Chapter 5

Compilation Method

Translation of an OCCAM description to a circuit involves three major steps. First each OCCAM language construct is replaced by a circuit that implements that construct. This is a syntax-directed replacement that replaces each statement in the program description with a corresponding circuit. Next the resulting initial circuit is improved using semantics-preserving circuit to circuit transformations. This involves identifying subcircuits that can be replaced with more efficient circuits without altering the semantics of the initial circuit. Finally the resulting circuit can be implemented by using a library of standard cells. The physical wiring of the circuit modules may be done by hand or automated using a variety of place and route tools.

5.1 Initial Circuit Replacement

The main step in translating OCCAM programs into circuits is the initial substitution of circuit structures for OCCAM language constructs. The most critical aspect of this initial syntax-directed circuit replacement is that the circuit must have the same behavior as the piece of OCCAM code that it is implementing.

Recall from Chapter 3 that one model of a simple OCCAM process is as a black box that begins execution after a request signal and the signals completion with an acknowledge signal. The schematic representation of this model of an OCCAM process was shown in Figure 3.1. Following this model, the initial circuit translation of an OCCAM process is a circuit or assembly with a request input to tell the process to start, an acknowledge output to report when it has finished, and possibly some data connection. The circuits that implement the processes of OCCAM are designed so that the composition of processes in the OCCAM source code corresponds to the composition of circuits in the translated source.

As with the process model, these circuits may be connected together and viewed as larger black boxes which use the same request-acknowledge control interface. By allowing these sets of modules or circuits to be viewed as a single process from the outside, the hierarchical structure of the system may be captured. The entire system being described by the OCCAM program may be viewed as one OCCAM process with a request to tell the process when to begin execution and an acknowledge to signal completion.

In order to start the operation of the system after initialization the environment issues a transition signal called *start* after a master clear to get things started. This signal is connected to the request input of the process obtained by translating the single top-level OCCAM process. When that top-level process has completed, it produces a transition signal on its acknowledge.

5.1.1 Translations of Primitive Processes

The three primitive processes of OCCAM from which all other processes are constructed are assignment to a variable, output to a channel, and input from a channel. The translations from these primitive processes into initial circuits are shown in Figures 5.1, 5.3, and 5.4 respectively. These translations may seem to be more complex than necessary for such simple primitives. This is because they are designed to cope with any situation that may arise in an OCCAM program. Optimization transformations, explained in Section 5.2 will often make substantial simplifications.

Assignment

A variable translates into a Register module to hold a value and a number of input pathways to that Register as shown in Figure 5.1. Corresponding to each statement in the OCCAM program that assigns a value to the variable is a data input and a request-acknowledge pair. An assignment process is started by a transition on the request signal which corresponds to a request for a data value to be written to the variable. This request becomes the *Ren* of the Enable module which enables the data value to the input of the register. When the *Aen* signal acknowledges that the data are available at the input to the register a transition is routed through the Call module to the request of the Register. The Call module is needed because this variable may be set at different times and from different places in the source code. Each instance in the static code of assignment to the same variable must have its own separate enabled path to the input of the Register.

When latching is finished, and the Register output data are valid everywhere they are used, the Register generates an acknowledge, which causes the call module to acknowledge. This is connected to the *Rdis* input of the Enable and signals that the data should not be enabled to the output, and finally generates an acknowledge signal to indicate completion of the assignment. Note the ellipsis in Figure 5.1. Because a variable may be set from different places in the same program, there are possibly many inputs to a single variable and thus many Enable modules, and equally as many inputs to the Call module. The ellipsis

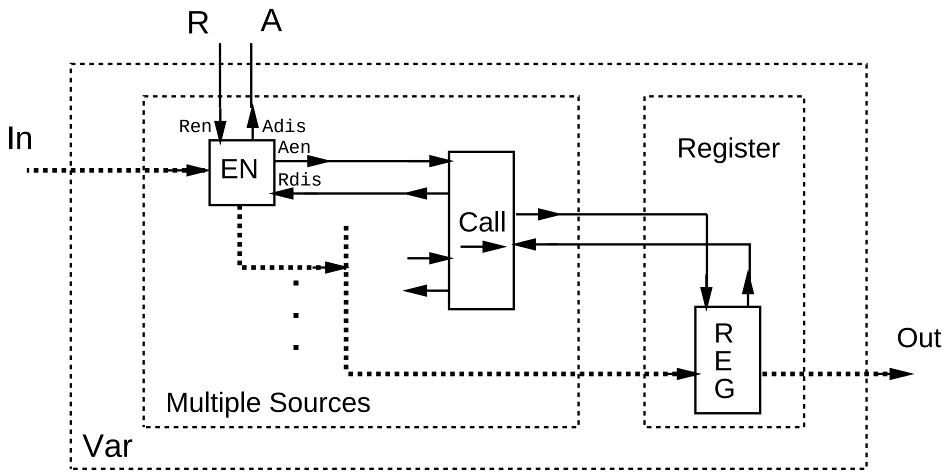

Figure 5.1: Initial Translation of Assignment Primitive

indicate that the circuit structure may be repeated if necessary. Ellipses in other figures denote similar situations.

This implementation of assignment must ensure that the bundling constraint on the output of the Register is obeyed. The output condition on the Register is designed to make sure that other modules that use the value in this Register will have their bundling constraints met. The acknowledge from the entire variable circuit is not allowed until after the value at the output of the Register has had time to propagate to wherever it is needed. If this condition is enforced at the output of the Register, then the bundling constraint at the input to any other variable will always be met. The input bundling constraint for an assignment is that the data must be available at the input to the Enable module before the *Ren* request to enable arrives. This is a constraint on the environment of the variable because the individual circuit cannot force this condition, but as long as the input comes from another circuit that enforces bundling at its output, this input bundling constraint will also be met.

The only effect of executing an assignment is that the state is changed to reflect the new value of the location that holds the variable's value.

Input and Output

The OCCAM output and input primitives are used to synchronize two processes, and possibly to send data from one to another. Only after one process has requested output to a channel and another has requested input from the same channel can the communication take place and the processes proceed. In its simplest form, when no data are passed and a channel is used by exactly one output statement and one input statement, the translation shown

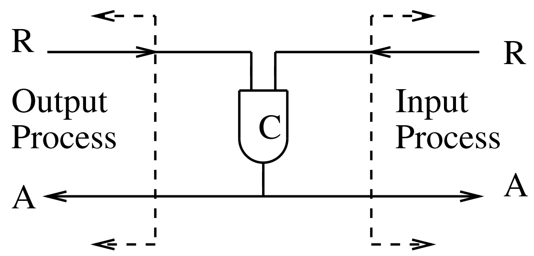

Figure 5.2: Synchronization in a Simple Communication Channel

in Figure 5.2 applies. The C-element provides the synchronization: when an output process has been started by its request and a corresponding input process by its request, the C-element output changes, thus signalling the completion of both the input and output processes.

Figures 5.3 and 5.4 show a more general implementation of a channel. This implementation allows data to be passed and allows multiple uses of the same channel by both output and input processes.

The output process, shown in Figure 5.3 is similar to the assignment process but without the register. Enabling circuitry is used to route data associated with different output statements to the same channel data bus. Note that the complete channel data path is simply a set of two control wires and a bundled data bus.

The C-element, shown in Figure 5.4, still controls the synchronization of the two processes. In the input process, a call module routes requests associated with different input statements to the C-element used for synchronization, and also arranges for the proper variable to be set. Depending on how the channel is used in the OCCAM code, many different variables may get their value from that same channel. Each variable, which is implemented as the circuit in Figure 5.1, will contribute an input to the Merge module that generates the single acknowledge for the channel. Note that there is a single C-element for synchronization regardless of the number of input or output processes using the same channel.

Figure 5.2 shows the complete symmetry of OCCAM synchronization. The question then arises whether the C-element should belong to the translation of the output process or the corresponding input process? When data paths are added, as in Figures 5.3 and 5.4, the symmetry is less evident, but the question remains. In this case the C-element is associated with the input process because this choice minimizes the number of wires associated with the channel. This choice also simplifies the implementation of guarded input commands, as will be seen below. Moreover, the channel itself is now exactly a bundled data path, with data wires, request, and acknowledge signal just as in Figure 4.4. The bundling constraint must be met for a channel the same as for any bundled data path.

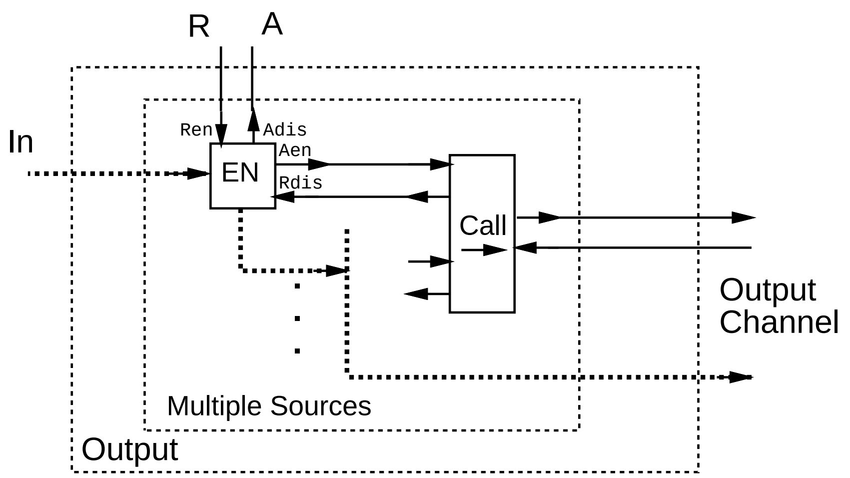

Figure 5.3: Initial Translation of Output Primitive

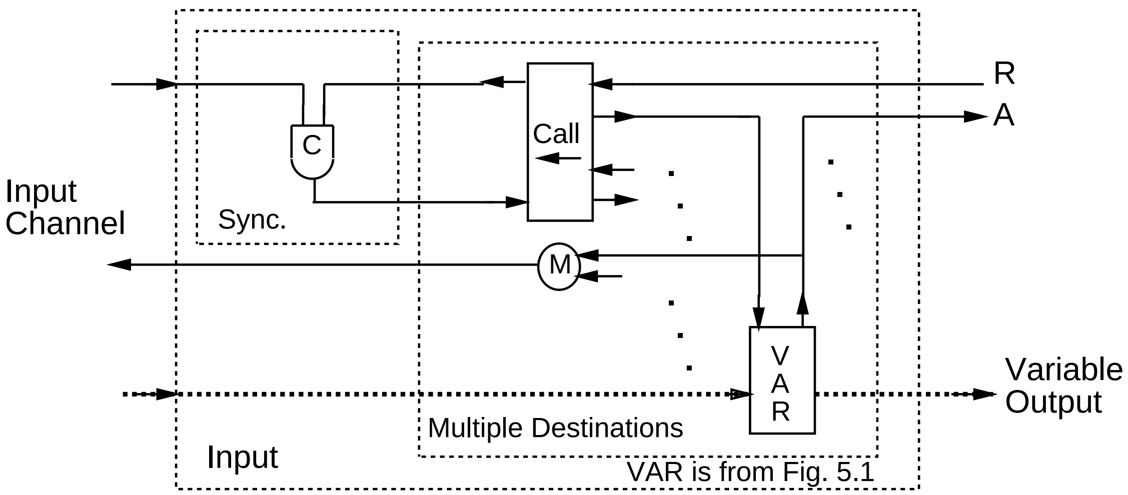

Figure 5.4: Initial Translation of Input Primitive

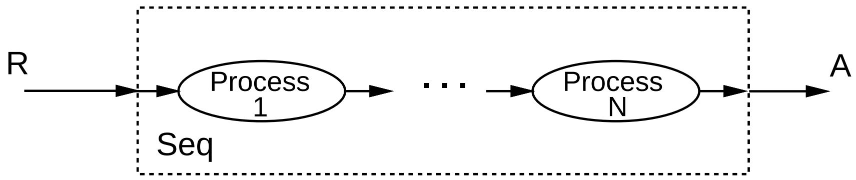

Figure 5.5: Circuit for a Sequential Construct

5.1.2 Translation of Process Constructs

OCCAM programs are formed by composition of primitive processes using the constructs described in Chapter 3. As with the primitive processes, each construct has a circuit replacement that matches the behavior of its language counterpart, and each initial circuit replacement may be improved significantly in the optimization phase.

Sequential Construct

The sequential construct simply chains the component processes together by connecting the acknowledge of one process to the request of the next process. In this way each of the n component processes, from 1 to N, is executed in sequence. The request to the construct is connected to the request of the first component, and the construct acknowledges when the last component process produces an acknowledge.

As before, the ellipses indicate that circuit structure shown may be repeated as necessary. Any number of processes may be executed in sequence.

Parallel Construct

The parallel construct uses a different connection of its component processes. A request signal starts each of the n component processes concurrently. The acknowledges of the components are collected in a C-element, which will produce a transition only after it receives a transition on each of its inputs. Thus the parallel construct will acknowledge only after each of its component processes has acknowledged.

As described in Chapter 3, communication is possible only between processes running in parallel. In Figure 5.6, for example, an input process contained in process 1 could receive data from an output process in process N. Communication is also possible with the environment which can be considered as another process that always runs in parallel with whatever system is being described.

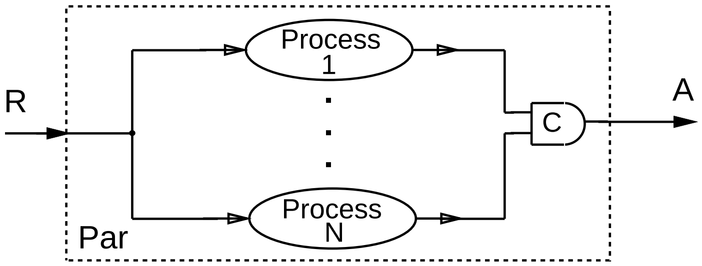

Figure 5.6: Circuit for a Parallel Construct

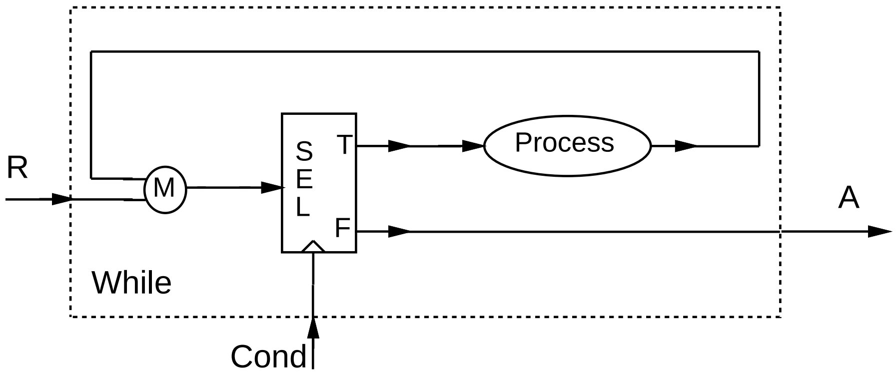

Figure 5.7: Circuit for a Repetitive Construct

Repetitive Construct

The repetitive construct implements a loop, which is terminated when the conditional expression is found to be false. A select module is used to test the condition. If the condition is true, the component process is started. When the process is finished the condition is tested again. If the result of testing the condition is ever false, the repetitive construct terminates and produces an acknowledge. Note the natural correspondence between the wiring diagram and the "flow of control" of the program.

The repetitive construct satisfies the bundling constraint required by its select module because of OCCAM's restrictions on the use of shared data. If the condition depends on variables that may change, then this change must either happen inside the process that is executed as the body of the *While* construct, or else the condition will not change at all while the repetitive construct is active. In the later case, the loop will either be non-terminating, or will not execute at all. OCCAM's rules about shared data ensure that no other process running concurrently may change the value of the condition while the repetitive statement is testing that condition. If the condition will change, then it must depend on variables that are changed by the process executed within the *While* loop. An assignment to a variable is

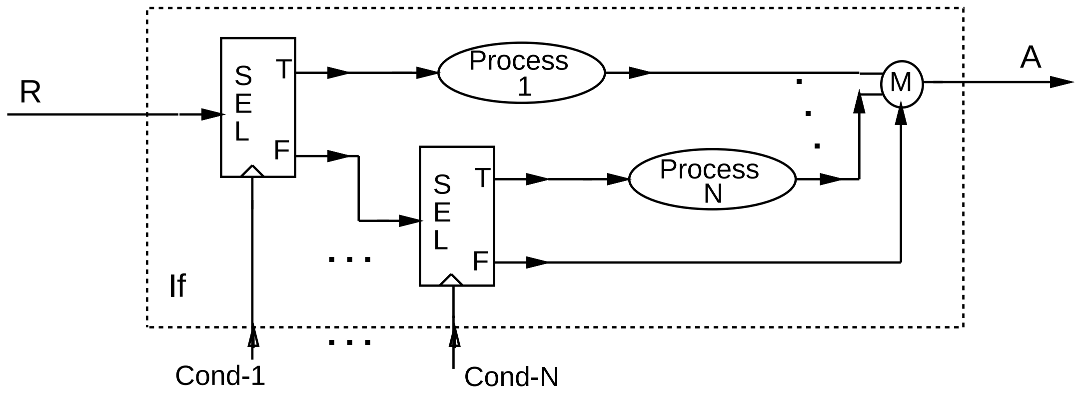

Figure 5.8: Circuit for a Conditional Construct

not allowed to terminate until the variable's data values have propagated through any circuit that uses the data. Thus the data value available to the select module is valid before the process embedded within the *While* completes. This observation is sufficient to establish the bundling constraint. If the conditions on shared data were relaxed such that the bundling constraint might not be met, the circuit translation for a repetitive construct would require using a Q-Select rather than a simple Select module.

Conditional Construct

The conditional construct also uses select modules to test the value of conditional expressions. The condition corresponding to each process is tested in turn. If the condition is true, the process is executed and the conditional construct terminates. If none of the conditional expressions is true then the construct acknowledges with no other effect. Note that this is different from CSP where the conditional construct stops if there is no condition that is true. Because at most one of the component processes will execute a Merge module is used to form an acknowledge for the entire construct.

Following the behavior of the conditional expression, the conditional clauses are tested sequentially in lexical order. This is implemented by physically wiring the selectors so that the lexical order is followed. As in other circuit diagrams, the ellipses indicate that any number of conditional expressions may be tested in an *If* construct.

Alternative Construct

The alternative construct is used to select one process from a set of guarded processes to execute. A guarded process consists of a process and an associated guard that must evaluate to true before the process may execute. The guard may include a Boolean expression like the expressions used by conditional and repetitive constructs, and may also include an input statement. In this case the guard is valid if the Boolean is true and there is input available on the given channel. If a process guarded by an input command is chosen for execution, the input process in the guard is completed before the component process is started. The alternative construct differs from the conditional construct in that while a conditional construct will execute *at most* one component process, an alternative construct will execute *exactly* one component process. The alternative construct will wait until a guard is true and then execute the process associated with that guard. When that process finishes, the alternative construct acknowledges. If more than one guard is valid, an alternative construct may choose any one of the valid guarded processes to execute. It is this choice that makes an alternative construct interesting to implement and it is the manner in which this choice is made that distinguishes one flavor of alternative construct from another. The following section presents a variety of different implementations of an alternative construct. These implementations use different techniques to decide which guard to check, and use different circuits to make a choice in the face of multiple valid guards.

The reason this sort of choice is interesting is that it is the mechanism used to select one request for communication from among several possibilities. Any process that communicates over two or more independent channels may run into this problem. In an OCCAM *Alt* construct, a guarded command may have a request for communication with the environment as part of its guard. For example, a guarded command with an input statement as part of the guard is ready to execute if there is input available at the given channel. This is similar to the *Probe* command proposed by Martin [56]. The main difference is that while Martin's *Probe* is an explicit function that produces a Boolean result that may be tested, an input guard in OCCAM is not tested explicitly and, according to the semantics of OCCAM, the communication must be completed before the body of that alternative is executed. This means that, unlike using the *Probe*, a program cannot delay the communication in a valid guard until after some other process has been completed. The process of testing the input guard to see if there is communication available entails completing that communication if the guard is valid before any other action is taken.

If a choice must be made between two or more independent requests for communication then at some point a non-deterministic choice between two or more external signals must be made. Because these external signals are not synchronized to the process that is making the choice any implementation must use a device like an arbiter to make the selection. The time it takes for an arbiter to reach a stable state if both its inputs are changing at the same time is potentially unbounded due to possible internal metastable states in the arbiter [16, 83]. Any implementation of an alternative construct must take this into account by using delay-insensitive control signals.

An additional complication in an OCCAM *Alt* construct is that there may be more than one valid guard when the alternative construct is executed. The semantics of *Alt* require that one of the valid guards is chosen, but does not specify how that choice is made. In fact, due to the behavior of the circuit sampling the guard input, the semantics of *Alt* cannot require that the choice is made in any particular way. Consider, for example, an implementation that attempts to guarantee that the valid guard with the highest priority will always be selected. This implementation can fail if the arbiter takes so long to consider a particular guard that a guard with higher priority has become valid in the meantime. The only guarantee that an alternative construct can make is that the guard that is selected is valid. Any ranking, or other ordering of the valid guards, is not possible.

The previous example gets at the heart of the problem. Any attempt to enforce some method of choosing from amongst a set of valid guards *must* introduce a notion of time into the choice. It is tempting to introduce time by modifying the semantics of *Alt* to require that the implementation always choose the guard that was valid first. Again, because of the properties of the sampling circuits, this is not possible. The notion of time that is introduced must be discrete. An implementation can sample all the guards at some point in time. After the sampling circuits have resolved, a choice can be made from amongst the guards that were found to be true during that sampling period. The next choice will be initiated with another sampling period. Changes in the status of a guard will be noticed only when the sampling begins. Further changes after the value of the guard has been resolved will not be noticed until the next sampling period.

This is similar to the familiar notion of a synchronous circuit using a clock to introduce a discrete notion of time to a system. Changes at the input to a register will not be noticed until the clock indicates that the inputs are to be latched into the register. Further changes to the inputs will not be noticed until the next clock signal. The main difference is that in a clocked system, the period of the sampling signals is fixed. In an *Alt* statement, the length of the sampling period cannot be fixed to a particular length because the sampling circuits may take an arbitrarily long time to exit from a metastable state. The requirements for an *Alt* construct are expressed in terms of the sequence of events rather than some absolute time interval. The choice of a valid guard can happen only after the sampling period, and the next sampling period may not begin until after this choice is made.

Using this notion of sampling the guards only during discrete periods, there are a variety of methods for choosing between the set of guards valid during that sampling period. Possibilities include choosing randomly from amongst the valid guards, choosing based on some priority ranking of the guarded processes, or trying to choose fairly so that under repeated invocation of the alternative construct each guarded process will have a fair chance of being selected. Note that this notion of priority is restricted to mean priority among the guards true after one sampling period. This is quite different from absolute priority which would be blind to when the sampling takes place.

A Transition Based Alternative Circuit The first implementation is one where the guard on each alternative is a transition signal. A transition on this guard wire will signal that the guard is valid and the associated process may be executed. For example, the guard on each alternative might contain a constant *True* Boolean part and an input statement for the

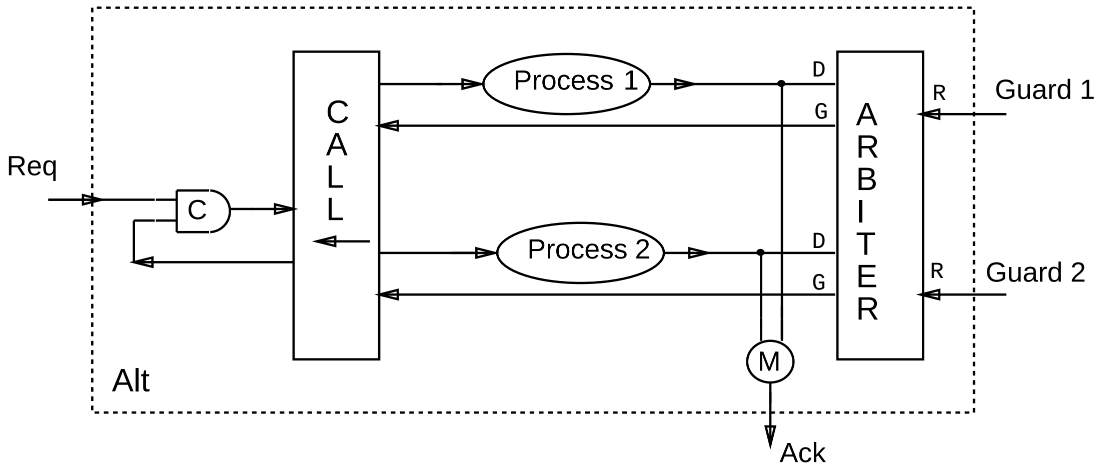

Figure 5.9: An *ALT* Module Using a Transition Arbiter

process part and thus be valid only if there is some input waiting at that channel. In this case the guard signal is the transition that comes from the channel signalling that there are valid data present at the channel. No notion of time is used in this circuit so no guarantees are made about which valid guard is eventually selected. The code for this might look like the following example:

```
(Alt  ((True (? chan1))      ; First alternative, Input guard on chan1
         (body1))            ; Body of first alternative
      ((True (? chan2))      ; Second alternative, Input guard on chan2
         (body2)))           ; Body of second alternative
```

The first instinct is to use a transition arbiter to make a choice between the guards. A possible circuit is shown in Figure 5.9. The Arbiter receives both transition guards as input requests, makes a choice between them, and causes a transition on the appropriate grant wire *G*. This enters the Call module as a request which sends out a transition to the C-element indicating that the Arbiter has selected a guard.

The *Alt* statement itself is started by a signal on the *Req* line which sends a transition to the C-element. The transitions may arrive at the C-element in any order, but no alternative will be executed until both input transitions have occurred. The C-element guarantees that only one alternative is selected for each transition on the *Req* line. When the transition appears at the output of the C-element the Call element acknowledges. This sends a request to execute the selected alternative's body process. Note that because the guards are input statements, each body processes shown in Figure 5.9 is actually two processes in sequence to correctly follow the semantics: first the input statement in the guard is executed, and then the body process is executed.

When the body process has acknowledged, this causes a transition on the appropriate *Done* wire of the arbiter. A transition on either *D* wire will send an acknowledge signal from the whole *Alt* statement, *Ack*, through the Merge. This also frees up the arbiter to make a new choice of guards and the whole process is ready to start over again.

The limitations of this implementation lie in the use of a transition arbiter to make the guard selection. Because the arbiter is a transition sensitive device, the guards that drive its request inputs must be transitions. If the guard is strictly an input guard, the input request from the channel, which is a transition, may be wired directly to the input of the arbiter. Note, however, that the definition of a guard requires that it have a Boolean part. This part may be a data signal and thus be defined by its logic level rather than a transition signal. If the Boolean signal is not constant, then it must be converted into a transition in order to use the circuit in Figure 5.9. In this case it is possible to convert the logic level to a transition using a Select module to steer a transition based on the value of the Boolean. The Boolean part of the guard is wired to the *sel* input of the Select module, and a request transition to the Select module can now test the value of the Boolean and produce a transition based on its value. The bundling constraint will be met because the Boolean part of the guard must be valid before the *Alt* is requested, but the value of the Boolean must now be tested explicitly on each invocation of the *Alt* statement.

A more serious problem is in the behavior of this *Alt* implementation if a channel that is used as the input guard in one alternative is used as a part of the body of a different alternative and that *Alt* statement may be executed more than once, say as part of a loop. In this case it is possible to violate the interface conditions of the arbiter which can cause unpredictable results. Code that illustrates this problem might look like the following:

```
(While cond
        ⋮
  (Alt  ((True (? chan1))      ;First alternative, Input guard on chan1
           (body1))            ;Body of first alternative
        ((True (? chan2))      ;Second alternative, Input guard on chan2
           (? chan1)))         ;Second alternative body, input from chan1
        ⋮
)
```

Imagine that there is input available on both *chan1* and *chan2* and thus both guards are valid and have sent a transition to the *Alt* statement. Because the Arbiter might choose either valid guard, assume it chooses the guard of the second alternative. The body of the second alternative now executes an input statement getting input from *chan1*. Remember that data were available at *chan1* so this input statement executes without waiting. Unfortunately, this sequence of actions has violated the environment of the arbiter in the *Alt* statement.

The request transition consumed by the arbiter from *chan1* is no longer valid and the interface specification of the transition arbiter used here does not allow a requesting process to withdraw that request before the request is granted. If the body of the loop is repeated and the *Alt* statement is executed again, the Arbiter in this particular implementation will be in an illegal state and results will be unpredictable.

The result is that this circuit cannot be used any time that a channel is used both as a guard of one alternative and in the body of another. In fact the problem is worse than that. The same trouble crops up if a channel is used as a guard to an *Alt* statement and also used for input anywhere else in the the entire program.

Note that this is not an intrinsic problem with any implementation of *Alt* using an Arbiter, but only a problem when this version of a transition Arbiter is used. A more general Arbiter could be built that would allow process to withdraw a request before that request is granted. This would allow the guard request to be taken away in the case where a channel is used outside the guard in the *Alt* statement and would avoid misusing the Arbiter.

A Ring-Style Alternative Circuit One reason for this problem with *Alt* statements that use a transition arbiter is the fact that the guard is a transition. The nature of a transition signal is that the change in logic level causes some reaction in the receiving circuit element. The use of transitions as signals dictates that once a transition occurs it cannot be "taken away." This is precisely the problem mentioned previously; a transition is sensed as an input guard to an alternative that is not chosen, and is later "revoked" by some other input statement communicating on the same channel. One answer is to use a guard that reports the possibility of communication as a logic level instead of a transition. This also means that the circuit must choose to look at the level to determine the validity of the guard instead of the guard signal itself driving the actions of the circuit.

Another way to look at this is that a transition arbiter has no notion of time and cannot choose when to look at its request inputs. If it is in a state where an input request is possible, then it must be receptive to that transition. A Select module, on the other hand, looks at the value of its *sel* input when it receives a request transition and steers that transition to an output based on that value. Changes in the *sel* input have no effect when the Select module is not requested.

If the input guard is required to be a level signal, then some means for generating a level signal is required because the request from the channel is a transition. The circuit in Figure 5.10 can be used to generate such a level signal. Recall from Section 4.1.1 that when a two phase handshake is in the state where the request is active but the acknowledge has not been seen, the two signal wires will be in opposite logical states. The XOR gate is used to detect this state by sensing that the two inputs of the synchronizing C-element differ. This indicates that either the input process is waiting for the output process, or vice versa. If the input process is part of a guarded command, it must be the case that the output process has signalled a request and is waiting for the input processes to complete the synchronization.

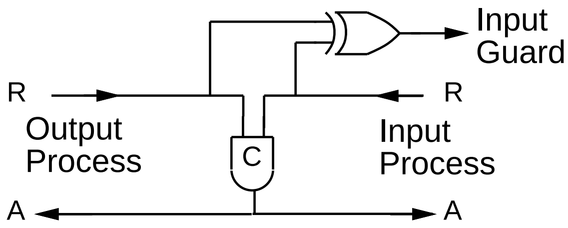

Figure 5.10: An Input Guard Signal

The signal produced by the XOR is just like a *Probe* of that channel; a high signal indicates that the partner is ready to synchronize. This signal is exactly what is needed for the *sel* input of a select module used to sense the guards validity.

Once the valid guard has been sensed, the input process may request input knowing that there will be input available at the channel. The request from the input process will reset the guard low by sending a transition to the other input of the XOR. Note, however, that this results in a new bundling constraint in the circuit. The request from the input process is delivered both to the input of the C-element and to the input of the XOR. The signal through the C-element must not be routed back to the circuit sensing the input guard until after the XOR has produced its output.

Note that the *Probe* signal generated by such a circuit may change at any time because the input process has no control over the output process, and therefore the signal does not satisfy the bundling constraint for a Select module. A Select module requires that the *sel* signal be bundled with respect to the request transition. This makes it unsuitable for implementing an alternative construct that uses an input process in its guard. This is exactly the same problem that required the use of an Arbiter in Figure 5.9; the guard may be changing at the same time that the alternative construct is sampling it. This problem is solved by using a Q-select module instead of a simple Select. Recall that a Q-select module uses a Q-flop to sample the *sel* signal upon receipt of a request. After the value has been reliably sampled, the Q-select produces a transition on one of its outputs.

If a level guard is used and a Q-Select is used to check that guard, then a simple way to implement an *Alt* statement is to check all the guards one after another until a valid guard is found. In fact, checking the guards one by one makes the *Alt* statement look something like a standard conditional construct. It checks some conditions one by one to determine where the flow of control goes next. There are some major differences however. The first is that if no alternative is ready to execute an alternative statement will wait until one is ready and then execute the corresponding process before exiting. If none of the conditions to a conditional statement evaluate to *True* then no other process is executed and the conditional construct exits. This means that the *Alt* construct will have to include some sort of loop to keep checking until a guard is found to be valid.

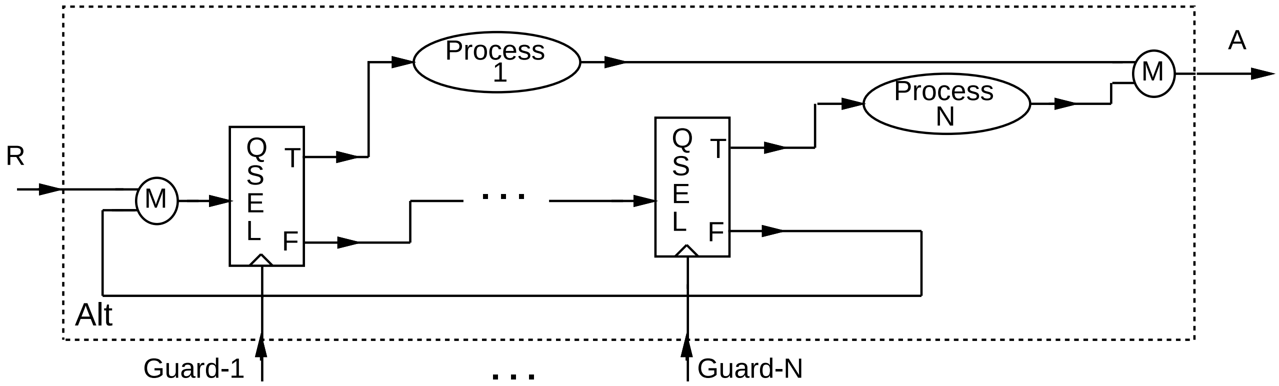

Figure 5.11: A Ring Style Alternative Circuit

Another difference is that, as mentioned previously, the guard in an *Alt* statement might be changing at the time it is being checked and thus must be sampled reliably. The conditions in a conditional statement must be Boolean and must be stable before the conditional statement is executed.

A circuit that implements an alternative construct of this sort is shown in Figure 5.11. The *Alt* is arranged as a ring of Q-Select modules that check the guards in a round-robin fashion. The *Alt* statement is initiated with the *Req* signal which starts the loop. The Q-Select modules will check each of the guards until one is found to be valid at which point the body process associated with that guard will be requested. When any of the body processes acknowledges then the alternative construct terminates and acknowledges on the *Ack* line. Because each guard is checked only during the time that the *Alt* statement is active, the problem associated with using the same channel in a guard and in a body is avoided. If a guard is true but not selected it does not matter that it might become invalid because some other input statement engages in the offered communication at another place in the code.

Because this implementation of an alternative construct contains a loop there is a simplified circuit that can be used if the *Alt* is the only process inside a *While* loop. For example, it is common to see an *Alt* statement as the body of the top level loop so that the system chooses and executes alternatives continuously. For example, the outline of the code may look like the following example:

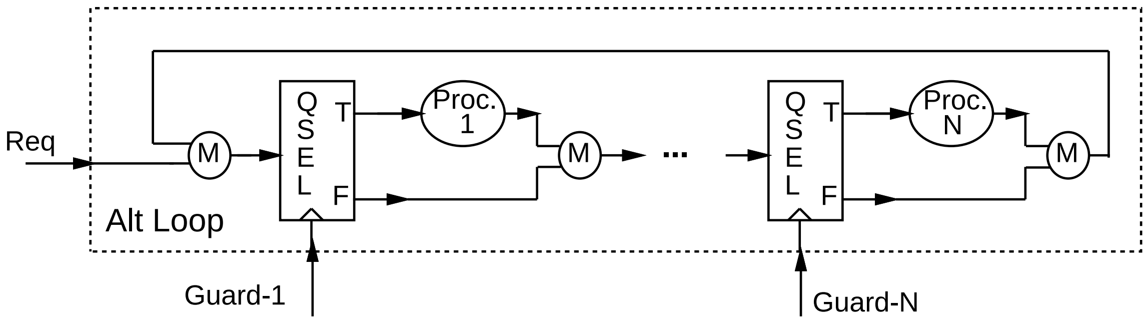

Figure 5.12: An Endless Loop Alternative Construct

```
(While True                          ;Execute forever...
   (Alt  ( alternative 1 )           ; choose one alternative and execute it
         ( alternative 2 )
             ⋮
         ( alternative n )))
```

The obvious solution in this case is to use the circuit from Figure 5.11 and simply to connect the *Ack* signal to the *Req* signal through a Merge module making an endless loop that executes the *Alt* statement continuously. In this case, however, another circuit option is available. If the *Alt* statement is the only statement in the top level loop then the action of the circuit will be to loop forever selecting valid guards and executing the associated processes. When a process completes, the *Alt* will select another valid guard and continue. In this case the outer loop may be eliminated completely, making the alternative construct itself the top level loop. An implementation of a circuit with this behavior is shown in Figure 5.12. Each guard is checked in turn and if the guard is valid then the associated process executes. After a process is finished the *Alt* loop continues its checking with the next alternative.

Note that this circuit will have a subtly different behavior than the circuit in Figure 5.11. In the former case the *Alt* statement will begin checking with the same alternative every time through the loop. That is, each time a body process completes, the *Alt* will begin checking for a valid guard with the first process again. The later circuit will begin checking with the next guard in the circuit. Because the semantics of *Alt* does not specify how the guard choice is made, either circuit meets the specification.

In particular, because the alternative construct is silent on the issue of how the choice is made, it makes no attempt to guarantee that the choice is *fair*. Fairness can be defined in many ways, but in general this means that there is no guarantee that any particular valid guard will ever actually be selected for execution.

A Lexical Alternative Circuit In the previous section the *Alt* implementation shown in Figure 5.9 will generally choose the first guard that becomes true, while the implementation in Figure 5.11 will check each guard sequentially in round-robin style. It is sometimes the case, however, that some alternatives are "more important" than others and should be given priority if their guards are valid. While true priority is difficult to implement, a simple way of expressing a limited form of priority is to define a form of alternative statement which checks guards in the order in which they are listed in the code. This way the most preferred alternative can be listed first in the code and will be checked first under repeated invocations of the alternative statement. This is known as a *Lex-Alt* statement and is an addition to the constructs in this version of OCCAM. The code looks like the following:

```
(Lex-Alt  ((guard 1)        ; First alternative, Check this guard first
             (body 1))      ; Body of first alternative
          ((guard 2)        ; Second alternative, Check this guard next
             (body 2))      ; Body of second alternative
             ⋮
          ((guard n)        ; Last alternative, Check this guard last
             (body n)))     ; generic body for last alternative
```

A ring style *Alt* statement like the one in Figure 5.11 implements exactly this function. When the alternative construct is requested with the *Req* signal the circuit starts checking with the first guard in the list. It will check guards in order until it finds one that is valid. Each time the *Alt* is encountered it will start checking again with the first alternative.

A "Fair" Alternative Circuit Another possible modification of a general alternative statement is to attempt to make a fair choice from amongst the valid guards on repeated invocation of the alternative statement. A fair choice in this sense means that if a guard is valid then it is guaranteed to be chosen as the command to execute after some finite number of other commands have been executed. This form of alternative statement is called *Fair-Alt*. The easiest way to implement fairness is to check the guards in a round-robin order so that every alternative's guard will be looked at once for each trip around the loop. This way if a guard is valid it will be selected after at most $n - 1$ other statements are executed if the loop contains n alternatives.

If the *Fair-Alt* is the top level loop then the circuit in Figure 5.12 can be used. Each guard is checked in turn and after a statement is executed all other guards are checked before that statement can execute again. Also, if a guard is valid then that statement will be chosen after at most *n* other alternatives execute in a loop of size *n*.

The ring style *Alt* in Figure 5.11, however, is not fair using this definition. Each time the alternative construct is initiated the circuit begins checking with the same alternative. A greedy alternative could be selected every time which would prevent any other alternative

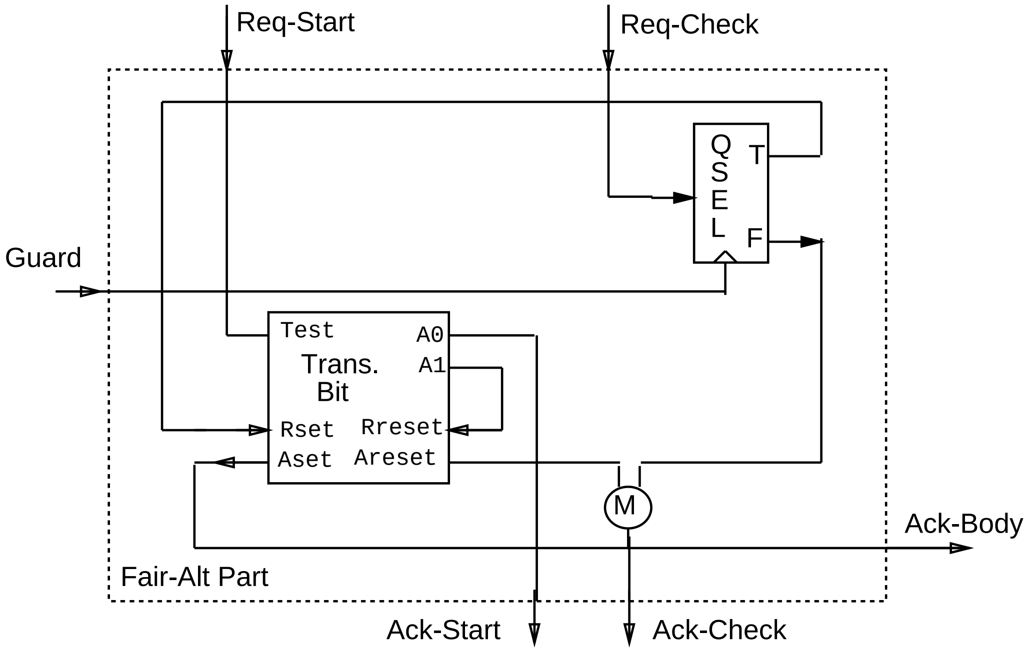

Figure 5.13: A Fair-Alt Part

from ever being selected for execution. If a circuit that is not the top level loop is to be fair then it must make sure that all other alternatives are checked before checking one choice twice. A ring style *Alt* is fair only if it is executed only once. If it is part of a larger loop and might be executed more than once then the *Alt* will not be fair.

In order for this style of *Alt* to be fair under repeated invocation, it needs to keep some state so that it when it is initiated it starts checking guards where it left off the last time that statement was executed. A circuit to implement this form of fairness might store this state as a set of flags implemented as asynchronous Boolean variables associated with each alternative. A circuit of this type for a single alternative is shown in Figure 5.13. These parts may be assembled into a ring style *Fair-Alt* module. This assembly of ring segments into a *Fair-Alt* module is shown in Figure 5.14.

The request to initiate the *Fair-Alt* first passes through the Boolean variables to choose where to start checking. As long as the alternative's flag is false the control passes to the next alternative without checking the guard. If the flag is true then the value of that flag is reset to false and guards are checked starting with the next alternative and continuing in a round-robin fashion until a valid guard is found. Once a valid guard is found and an alternative is selected, the flag is set to true before the body of the alternative is executed. This means that the next time the *Fair-Alt* is requested, it will start checking guards at the next statement in the loop. In this way every guard will be looked at once before any guard

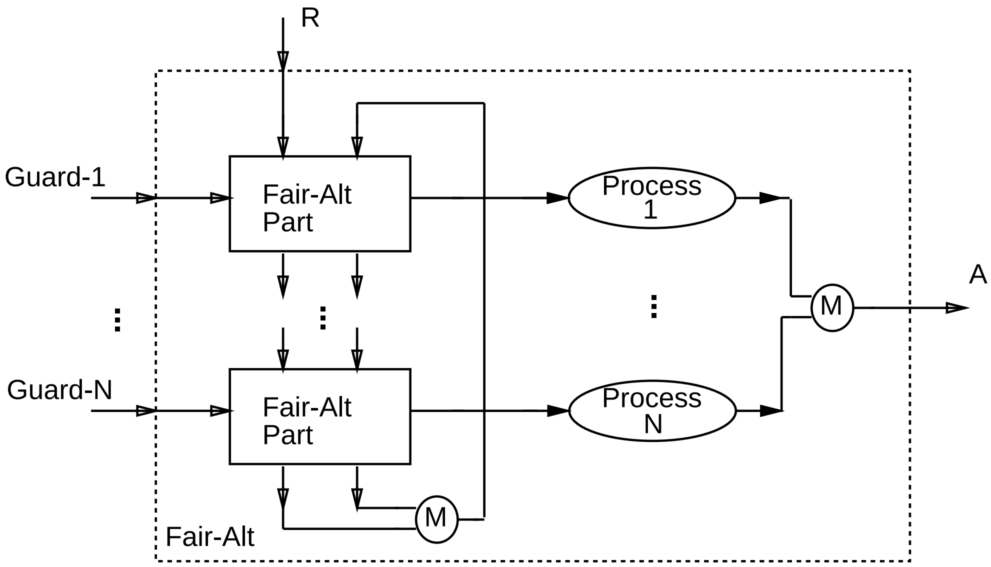

Figure 5.14: A Fair Ring Style *ALT* Module

is checked twice.

The Merge module at the bottom of the loop will ensure that in the case that no flag is set, as in the first time the *Fair-Alt* is requested, that the process of checking the guards will begin. In this case the guards will be checked starting with the first alternative's guard.

5.2 Circuit Improvement

Traditional software compilers use a variety of optimization techniques to improve the quality of the code they produce. This circuit compilation method also applies many optimizations during the translation process to modify the performance, size, or some other metric of the target representation while preserving the semantics of the original. Once the initial translation from program constructs to circuit constructs has been carried out, a variety of transformations may be applied to the resulting circuit. These optimizations are similar to peephole optimizations used in software compilers where improvements are made by looking at small pieces of code in limited contexts. A circuit-to-circuit transformation is described as an initial circuit and a replacement circuit. When a circuit topology is matched with a template it is replaced with a new structure that retains the behavior of the original but improves its performance in some way.

Circuit transformations must preserve circuit semantics. Recall from Chapter 4 that circuit semantics are based on trace sets. If a circuit replacement has the identical trace set as the original circuit at its external interface then the circuit semantics must be identical and the circuit replacement will be safe under any circumstances. It is the sequence of events at the interface that is important rather than the timing of those events relative to some system clock.

Although it is obvious that a circuit replacement may be performed if the replacement circuit performs exactly the same function as the original circuit, it is also permissible for a replacement circuit to be more general than the circuit it replaces. Dill calls this *conformation* [24]. A circuit conforms to another circuit if it allows all the circuit behaviors of the original circuit. A conforming replacement circuit may behave differently than the circuit it replaces but only in situations that the original circuit does not excite. This is similar to a situation where a replacement part can be used if it has better specifications than called for. For example, a home appliance that operates on a 110v supply can be replaced with an identical appliance that can operate on either a 110v or 220v power supply because the extra functionality is not used. In the realm of circuits, a circuit that uses a particular microprocessor will operate just as well with another microprocessor that provides a superset of the original instruction set. The extra functionality of the more capable part is not used in the design, but neither is it a problem because the extra instructions are never used.

It is essential to have a model for the possible actions from the environment in order to use a conforming transformation. This is to make sure that the replacement circuit behaves like the original in any situation that the environment might stimulate. In this case, the surrounding circuit provides the environment in which the replacement must operate, and moreover provides a specification for the behavior of the replacement circuit. To prove that the replacement is correct, a trace structure of the original circuit is developed and the replacement is proved to conform to it. This proof can be checked using an automatic tool to check whether the trace sets conform to each other [24]. This tool has been used to check the correctness of each of the transformations applied in this optimization process.

This proof establishes *safety* properties; a separate argument must be used to establish *liveness* properties, although the most common case of liveness problems, a module that does nothing and therefore conforms to everything, is eliminated by making sure that the circuit modules contain circuitry that will respond to input signals. That is, the module circuitry has active states where once activated the circuitry will respond to inputs by producing outputs. If the circuit modules themselves are live, that is, they will respond to inputs by producing outputs, then liveness for the composite circuit is not a problem unless the algorithm itself produces deadlock.

There are endless possibilities for circuit optimizations. The following sections show some common situations that arise in translating OCCAM programs, but are by no means complete. These examples are grouped into categories based on the types of modules that are optimized and are meant as examples of the types of transformations that can be applied.

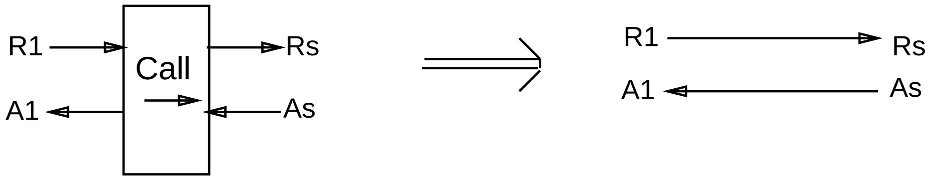

Figure 5.15: Removing a Single-Client Call Module

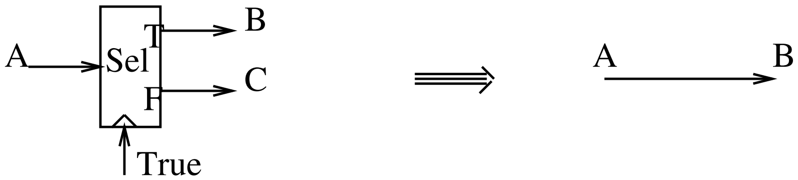

Figure 5.16: Reduction of a Select with a Constant Input

5.2.1 Null Module Elimination

The simplest transformations remove modules when the module's functionality is not used. Although simple, these transformations are very powerful and remove a great deal of unnecessary circuitry. For example, a Call module with only a single client may be replaced with a pair of wires connecting *R1* to *Rs*, and *As* to *A1* as shown in Figure 5.15. Note that the trace set of a pair of wires is different than for a single input Call, but when the environmental constraints present in the I-net of the Call are applied to the wire pair, the trace sets are identical. The environment is not allowed to make another transition on *R1* until after it sees a transition on *A1*, and the subroutine interface of the call is also required to follow the request-acknowledge rules shown in Chapter 4. If the environment follows these rules, which it must, the transformation is correct and may be applied. Anywhere that a single client Call module could be used, a pair of wires can be used just as well and with far less circuitry.

Similar null module reductions may be applied to Select modules that are instantiated with constant *True* or *False* conditions on their *sel* inputs. For example, in Figure 5.16 a Select module with a constant *True* on its *sel* input will always route its input transition to the *Tout* output. There is no need to include extra circuitry when every input transition is routed to the same output. A simple wire can be used instead. Similar transformations may be applied to Select modules with *False sel* inputs, and to Q-Select modules with constant *sel* inputs.

Another transformation on Select modules is possible if the *sel* input is driven by an inverter. In this case, the inverter may be eliminated, and the *Tout* and *Fout* signals of the Select module are reversed. The functionality is exactly the same, but the inverter has been eliminated.

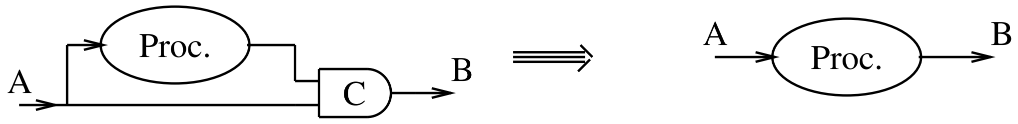

Figure 5.17: Redundant C-Element Reduction

```
(Par ((Chan Req Ack))
   (Seq ()                                    ;Master process
      (Do Process A)                          ;Execute some process A
      (! Req)                                 ;Request the slave process to do something
      (? Ack)                                 ;Wait for acknowledge from slave
      (Do Process B))                         ;Continue with some process B

   (Seq ()                                    ;Slave process
      (? Req)                                 ;Wait for request from master
      (Do Process C)                          ;Do some process C
      (! Ack)))                               ;Acknowledge back to master
```

Figure 5.18: Passing Sequential Control From Process to Process

5.2.2 C-Element Elimination

An extremely common situation regarding C-elements is shown in Figure 5.17. This situation usually arises when there are sequential sections in the code that synchronize through a channel to other sequential code segments. Whenever this situation arises, the C-element may be eliminated. If a box is placed around the entire circuit structure, the trace set of the interface is simply a standard request-acknowledge interface on *A* and *B*. The C-element is not synchronizing anything because the process must see the *A* input before starting, and the C-element cannot fire until after the process has completed. The C-element may be eliminated and the trace set of the interface remains the same.

A particularly interesting application of this type of optimization is when channels are used to pass control sequentially from one process to another. For example, the code for two processes might look like the code in Figure 5.18.

In this case the master process passes control to the slave process by signalling on the *Req* channel. The master then waits for the slave to send a signal on the *Ack* process before continuing. The master has passed control to the slave and is waiting for the slave to pass that control back. This is exactly the situation in a standard sequential composition except

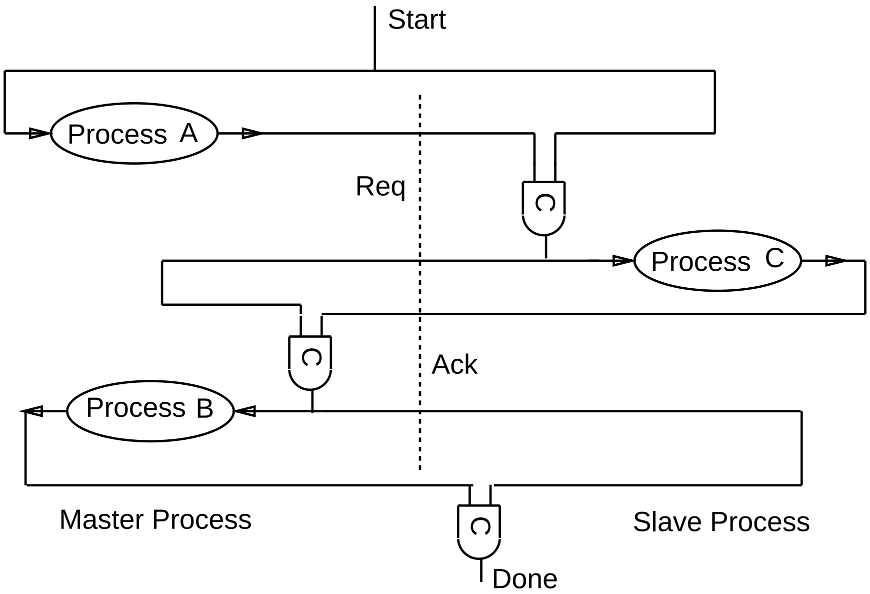

Figure 5.19: Initial Translation of Master-Slave Circuit

that because channels are used, there are four wires and two C-elements in the interface between the processes rather than a simple two-wire request-acknowledge interface. That is, rather than a single wire for the *Req* signal, a channel is used instead. The circuit is shown in Figure 5.19.

Application of C-element elimination optimizations can remove the excess circuitry and transform the circuit to that shown in Figure 5.20. Because control is passed sequentially between the processes, the channels have been reduced in this case to single wires. These wires are exactly what would have resulted if the code had been a simple sequential composition rather than one process passing control to another process. Although the code is written differently, the optimizations will reduce the circuit to a simple two-wire request-acknowledge interface.

5.2.3 Call Reductions

Call elements are introduced in the initial circuit translation whenever multiple uses of a shared resource, such as a channel or variable, are made. In many situations, however, the Call module is used in a way that can be optimized. For example, if multiple clients use a Call and then proceed to do some other, identical, action, the circuit will end up with multiple client acknowledgments connected to the same Merge module as shown in Figure 5.21. In this case, the size of the call may be reduced because the outputs are merged into a single signal by the Merge module. The fact that the Call steers the acknowledgment from the

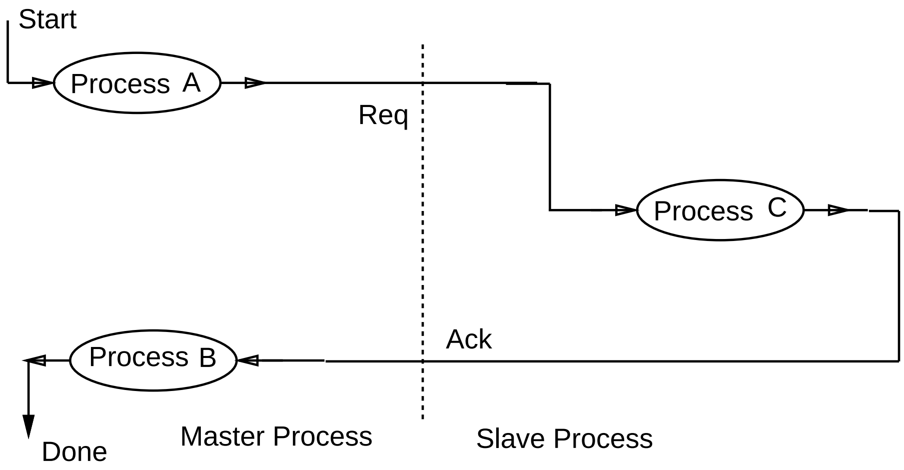

Figure 5.20: Optimized Master-Slave Circuit

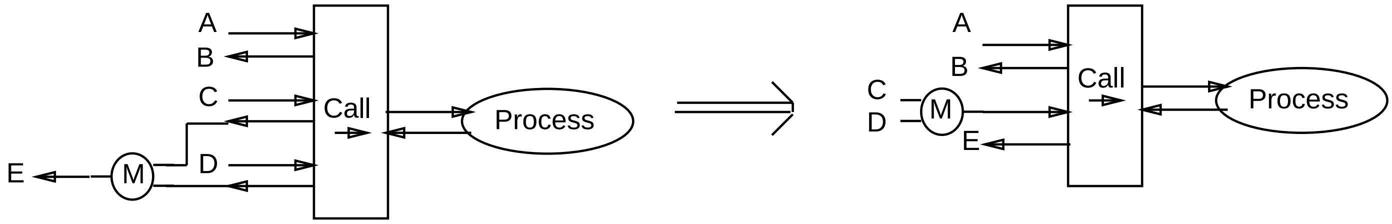

Figure 5.21: Call Size Reduction From Merge

called process to whomever called that process is irrelevant because both acknowledgments are immediately merged together. This is similar to the C-element situation where the trace set of the larger circuit is identical to the replacement circuit at the interface and the internal circuitry is reduced. Note that in the case that the Call module had only two requests to begin with, the call module is reduced to a single-client call and is removed completely by replacing a single client call with wires as described earlier.

A similar size reduction on a Call is possible in the case where every acknowledge of a call goes to another instance of a Call. In this case, as shown in Figure 5.22, the size of the second Call module can be reduced. If the next action after all the calls on a subroutine is to call another subroutine, then that second call need not be made by separate signals for each client. As before, the trace sets at the interface are identical, only the internal signals have changed, and if the size of the Call module is reduced to one client, that module may be eliminated.

Another way in which the size of a Call module may be reduced is if both outputs of a Select module are client requests of the same call module. In this case, shown in

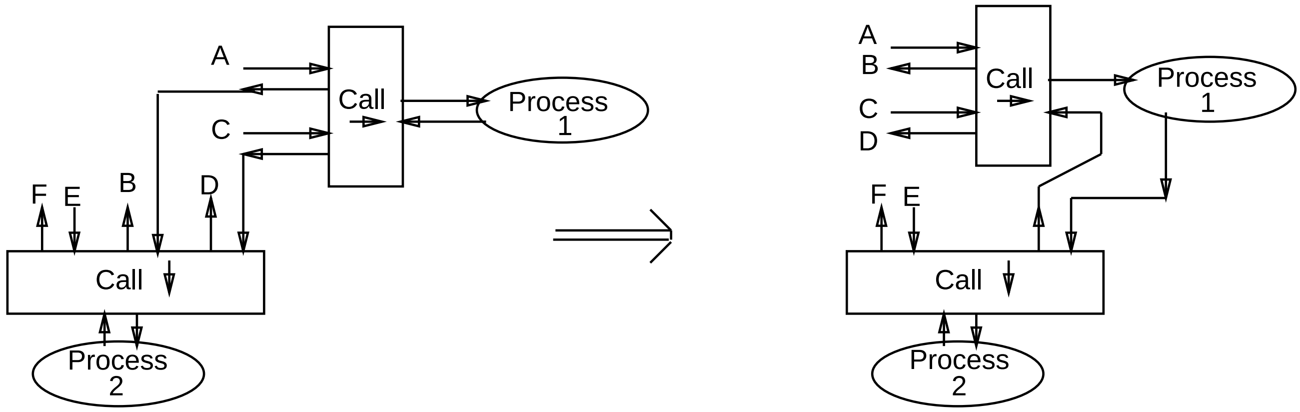

Figure 5.22: Call Size Reduction from Multiple Calls

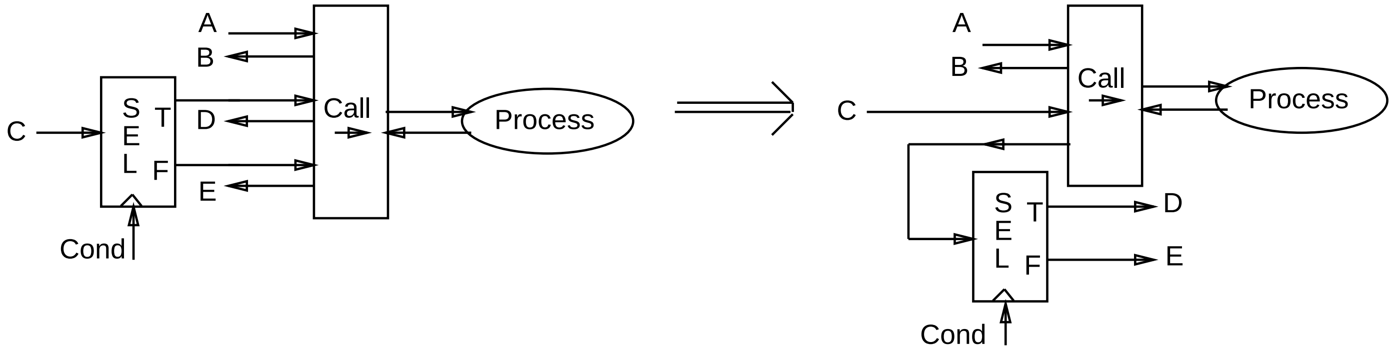

Figure 5.23: Call Movement Around a Select Module

Figure 5.23, the Call module *may* be able to be moved before the Select module. Note that this optimization must be applied with great care because the called subroutine may involve modifying the Boolean value tested by the Select module. If this is the case, the optimization cannot be applied because the order of the events in the circuit will be changed. In the original circuit the selection was made before the subroutine process was called. In the optimized case, the order is reversed and the process is executed before the selection is made.

In many cases, however, the called process does not directly influence the tested variable and the optimization can be applied safely. For example, the called process may be an output process to some channel and thus not have any effect on the current state of the Select module.

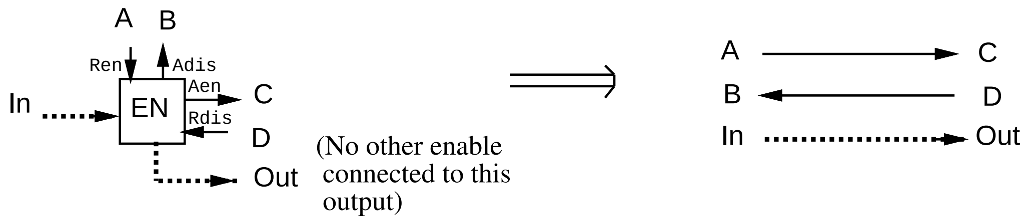

Figure 5.24: Removal of a Solitary Enable

5.2.4 Enable Reductions

Enable modules are designed to gate different signals onto a common bus. In particular, they will drive drive signals onto a shared bus upon request and then report that the values on the bus are correct. Given a request to disable, the module will stop driving the bus and make sure the module outputs are in a high-impedance state before reporting that the bus is available. Individual Enable modules are used to create multiplexers allowing data to be written into a single register from a number of sources. There are many situations, however, where Enable modules are not actually needed once the final circuit is assembled.

In the case shown in Figure 5.24, for example, there is a single Enable module connected to the bus. In this case, there will never be any other data values driven on the bus, and the Enable is superfluous and may be removed completely. It may be replaced with wires connecting *Ren* to *Aen*, *Rdis* to *Adis*, and *data-in* to *data-out*. As with all circuit modifications involving enables, the bundling constraint must be maintained after the circuit is changed.

In a more general case, Enable modules that gate the same data path onto the shared bus may be merged into a single Enable module and a Call module to keep track of which control path is requesting the service. This optimization is shown in Figure 5.25. Each of the individual requests to drive the data onto the bus must be mutually exclusive because they drive the inputs to a Call module, therefore, moving that Call module to the other side of the common Enable module does not change the semantics of the assembly. This optimization should also be applied with care because it reduces the size of one Call but adds a new Call module to the circuit. Depending on the width of the data path involved, there may not be any great decrease in circuit size after this optimization is applied, although in most cases there is a savings.

Finally, if a situation is discovered where a variable, like that shown in Figure 5.1, is of width 1 and set only with constant *True* and *False* values as the data inputs of the Enable modules, this assembly may be reduced to an asynchronous set-reset bit with Call modules to steer the control paths to the proper places. This optimization is shown in Figure 5.26.

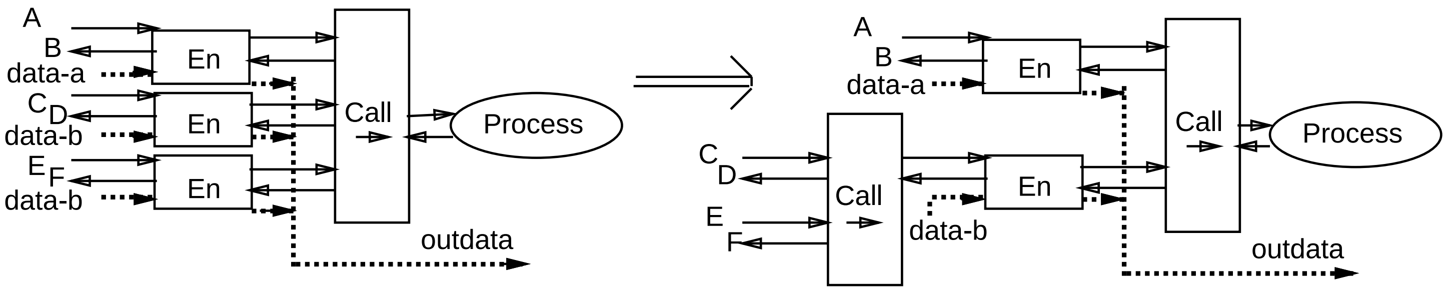

Figure 5.25: Merging Common Enable Modules

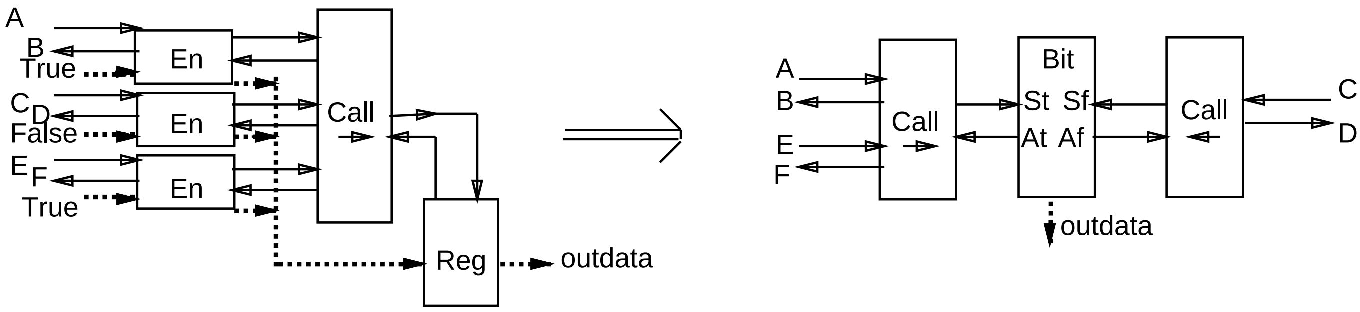

Figure 5.26: Single-Bit Constant Enable Reduction

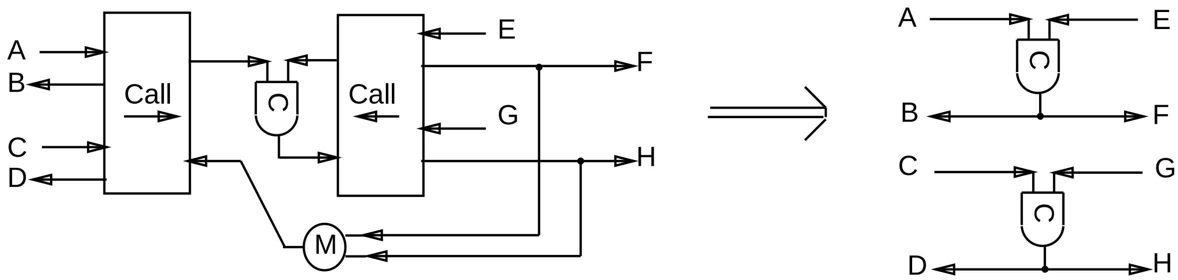

Figure 5.27: Channel Splitting

5.2.5 Channel Reduction

As a slightly higher level optimization, a single channel is often used for multiple purposes. For example, a single input channel may be used to receive both a control byte and the following data bytes in some sort of communication protocol. When these programs are compiled into circuits, however, multiple use of a single channel may use more circuitry than splitting the use into many different channels. This is because a Call module is required whenever the same channel is used in more than one place. If the calls on the channels are distinct in both the receiver and the sender, the channel may be split up into distinct channels. This is not an easy optimization to apply at the circuit level because it is often impossible to discern that the calls on the shared channel will always correspond to particular calls on the other end of the channel. Therefore, this is an optimization that should be applied by the programmer at the source code level. If there are distinctly different uses for the same channel, then that channel might better be split into multiple channels. The tradeoff is that multiple channels use more wires than a single channel so in cases like off-chip communication, where pins are at a premium, this optimization might not be desirable. The programmer has direct control over communication costs and thus should structure the program with those costs in mind. An example of the result of splitting a channel is shown in Figure 5.27.

5.2.6 *Start* Signal Elimination

A common and important transformation can remove the *Start* signal from the circuit in certain cases. Recall that *Start* is a transition that is issued by the environment after initialization to initiate action in the circuit. If *Start* is wired to a Merge module, for example, the Merge module produces an output transition on the upon receipt of a *Start* transition. In this case it may be possible to remove the *Start* signal and replace the Merge module with an inverter as shown in Figure 5.28. This transformation depends on the fact that modules respond to the *master-clear* signal by setting their control signal outputs

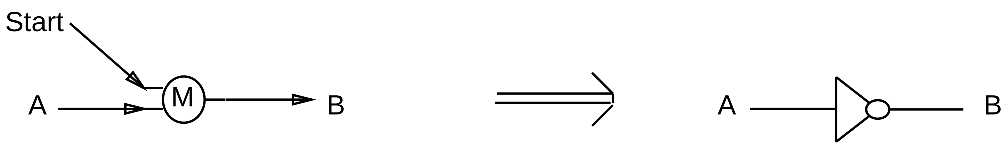

Figure 5.28: Removal of *Start* Signal from a Merge

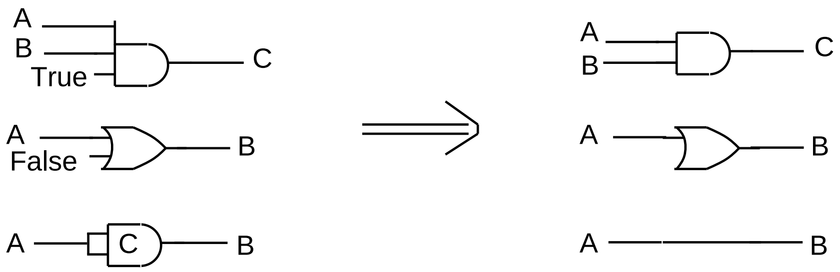

Figure 5.29: Example Gate Reduction Transformations

low, so the inverter will simulate the effect of a *start* transition issued immediately after *master-clear* is removed. This transformation is not always advisable, because in certain circumstances different modules will respond at different rates to the removal of the *master-clear* signal, and in certain circumstances chaos can ensue. In many cases, however, such as the FIFO buffer illustrated in the next section, the transformation is harmless because the *start* signal merely prepares the circuit for action, and does not initiate action.

5.2.7 Gate reduction

Simple transformations on logic gates are familiar to every designer of digital logic. Gates are used to construct Boolean expressions or in the transformation of language constructs. These optimizations all preserve exactly the trace sets of the gate interfaces and are easy to identify and apply. Some common logic gate reductions that occur in the translated circuits are shown in Figure 5.29. An *AND* gate with a constant *True* input may be replaced by a gate with that constant input removed. Likewise, an *Or* gate makes no use of a constant *False* input. Gates with a single input may be replaced with wires or simple inverters depending on whether the gate has an inverting output, and gates with identical inputs may be merged into a single gate.

```
(Process buffer ((Chan A B))                      ; Define a simple buffer
   (While True                                    ; Repeat forever
      (Seq ((Var temp<8>))                        ; Sequential composition
         (? A temp)                               ; get value from channel A, store in temp
         (! B temp))))                            ; send value in temp to channel B

(Block ((Chan input<8> output<8>))
   (Par ((Chan mid<8>))                           ; Parallel composition
      (buffer input mid)                          ; of two single-place
      (buffer mid output))                        ; buffers
```

Figure 5.30: Code for a Two Place Fifo Buffer

5.3 Example Compilations, Step-by-step

Perhaps the best way to understand the translation and optimization procedure described in this chapter is by looking at some examples. The following two examples show in detail the translation steps starting with the OCCAM source and proceeding through the initial translation to a circuit, and optimization of that circuit. The result is a circuit suitable for implementation.

5.3.1 A First-In First-Out Buffer

The two-place first-in first-out (FIFO) buffer shown again in Figure 5.30 is the first example. This example of OCCAM code was seen before in Chapter 3. The initial translation of the OCCAM program for the single *buffer* macro is shown in Figure 5.31. First the repetitive construct is expanded following the translation in Figure 5.7. The component process of the repetitive construct is a sequential construct with two component processes, an input and an output statement. Each of these processes is translated into an initial circuit. The dashed boxes in Figure 5.31 show the translation for each of the parts of the buffer macro.

Notice that the buffer macro begins operation upon receipt of a *Start* signal from the environment and signals completion with an *Ack* signal. There are two channels, one input and one output. The C-element that performs the synchronization at the input channel is included as part of the circuit for the input channel. The corresponding C-element for the output channel is included in the input circuit of whatever module this circuit is connected to.

Connecting two of these single buffers together to form a two-deep FIFO buffer is shown in Figure 5.32. This shows the transformation of the parallel construct with two

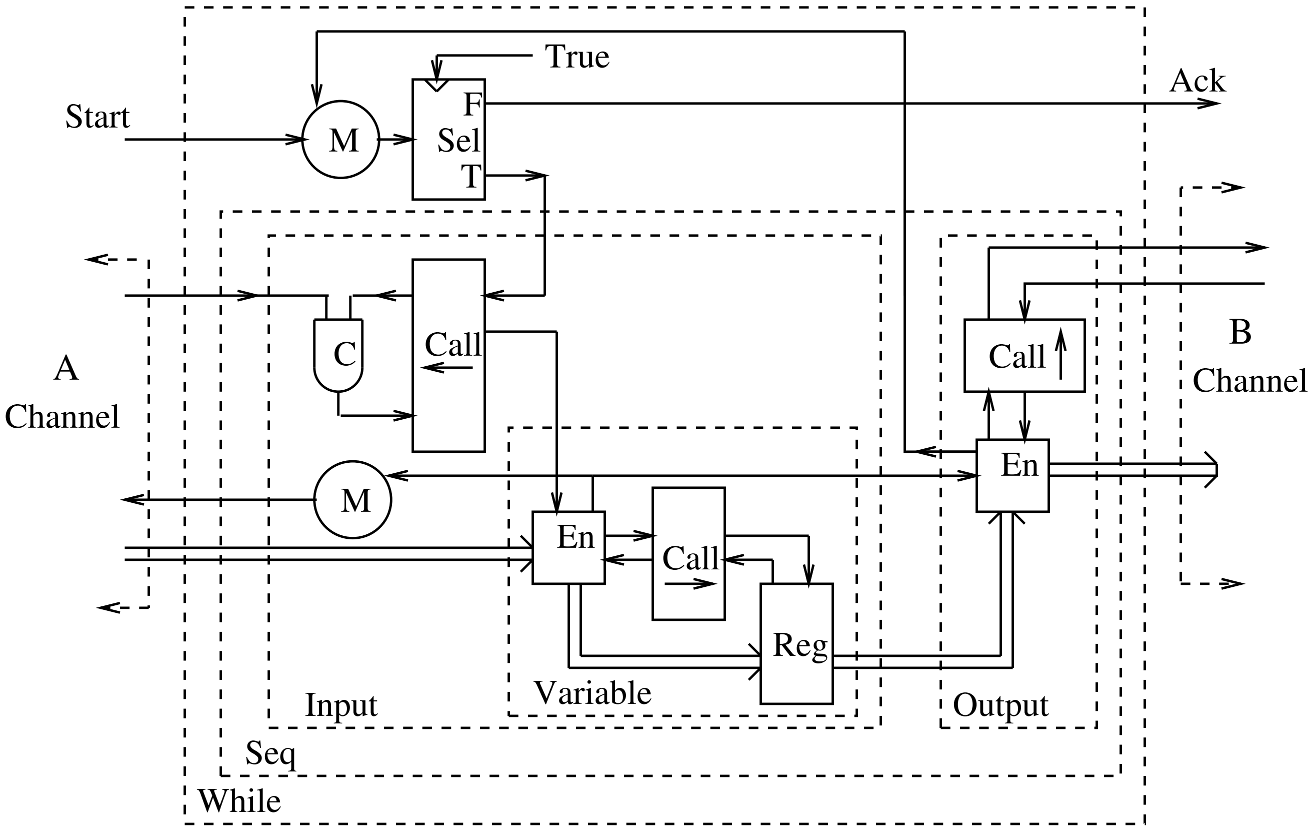

Figure 5.31: Initial Transformation of Buffer Macro

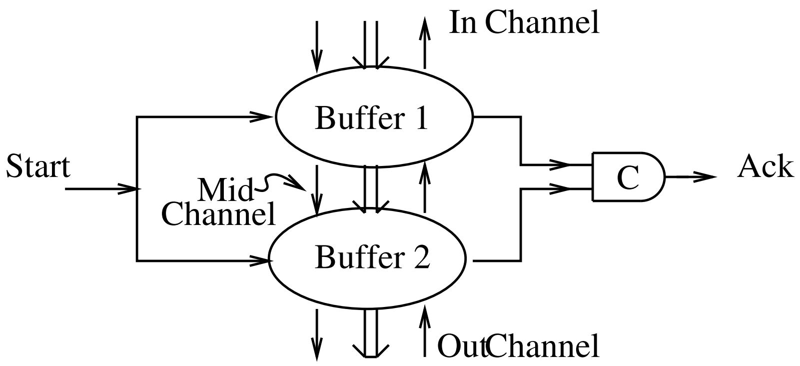

Figure 5.32: Transformation of Two-Place Fifo Buffer

component processes each of which is a buffer process.

Optimizing the circuit removes a great many components to yield the final result shown in Figure 5.33. Most of the optimizations are readily apparent in Figure 5.31: a Call module with only a single client can be removed and replaced by wires; an Enable module whose outputs are the only drivers of a data bus can be removed; a Merge module with one input can be removed; a Select module whose select condition is *True* can be removed. Note that when the Select module is removed, so is the acknowledge signal from the buffer macro. Because the main loop of the buffer macro is a non-terminating (*while True* $\cdots$) loop, the acknowledge is optimized away and does not appear in the final circuit. Once started, the FIFO will operate without terminating and never produce an acknowledge.

These steps, plus the removal of the *Start* signal, yield the final circuit. This circuit, which turns out to be the same as the FIFO circuit shown in [94], is the best circuit known for a FIFO using bundled data paths and transition signalling.

5.3.2 Distributed Mutual Exclusion

Whenever more than one independent process desires exclusive access to a shared resource, some sort of arbitration must be used. As explained in Chapter 4, the purpose of an arbiter is to make sure that only one of the competing processes has access to the shared resource at any time. A simple arbiter grants access to the shared resource to exactly one of two contenders at a time. The contending processes communicate with the arbiter using three signals as shown in Figure 3.5 and repeated here in Figure 5.34. Each contending process may signal on its *Req* line that it wants access to the resource. The arbiter responds on the *Grant* line to exactly one of the contending processes indicating that the process has access and may use the resource. When that process is finished it releases the arbiter with a signal on the *Done* line so that the arbiter may consider other requests.

Rather than have each of the processes communicate with a single arbiter process, it is

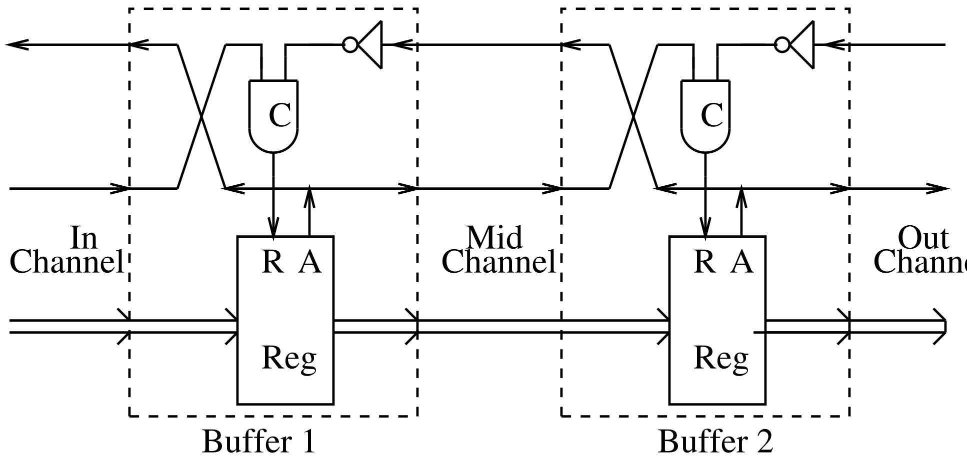

Figure 5.33: Optimized Two-Place Fifo Buffer

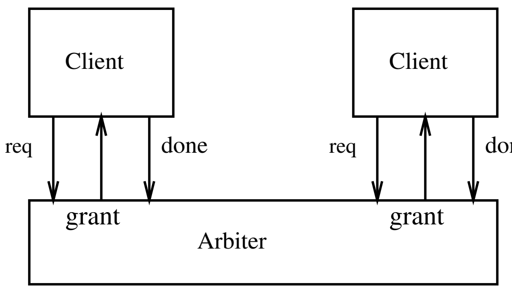

Figure 5.34: A Simple Two Input Arbiter (Same as Figure 3.5)

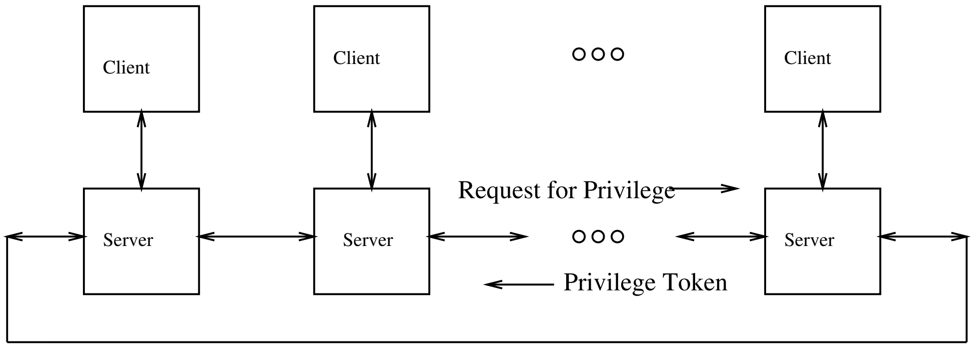

Figure 5.35: An Arbiter Distributed in Multiple Server Processes (Same as Figure 3.6)

sometimes desirable to distribute the arbitration amongst the contending processes. In this case, each contending process, called a *client*, communicates with a local process called a *server* that grants access to the shared resource. The server processes communicate between themselves to guarantee that only one client is granted access at any time. This idea was seen in Chapter 3 where a particular algorithm for implementing this form of mutual exclusion due to Martin [53, 55] was used as an example of OCCAM code.

In this section, two different algorithms for implementing distributed mutual exclusion are used to demonstrate the compilation method. First Martin's algorithm, seen in Chapter 3, is translated into a circuit, and then another algorithm for distributed mutual exclusion is described as another OCCAM program and translated into a different circuit.

Distributed Mutual Exclusion — Version 1

OCCAM **Source Code** In this example, the OCCAM code for distributed arbitration is based directly on the algorithm explored by Martin in [55]. The idea is that the server communicates to the master process on a channel called *Serv* and with the other servers over channels called *Right* and *Left* as seen in Figure 3.6 and repeated here in Figure 5.35. The requests for the privilege pass around the ring to the right, and the privilege itself passes around the ring to the left. Because only one server holds the privilege at any time, this guarantees mutually exclusive access of the master processes to the shared resource. Note that exactly one of the servers in the ring should be modified such that the privilege variable is true after a global clear signal. The OCCAM code for the server is shown in Figure 5.36 and the code that the master process uses to communicate to the server is shown in Figure 5.37.

Notice that the code contains two consecutive outputs to the right channel. Because no expression is mentioned, no data is passed and this communication reduces to synchronization. The first synchronization signals to the server on the right that the current server wants the privilege. The second synchronization means that the server on the right has passed

```
(Process server1 ((Chan Serv Right Left))
  (Seq ((Var priv))                       ; Define local variable.
    (While True                           ; Repeat forever.
      (Alt                                ; choose an alternative
        ((True (? Serv))                  ; Request from master.
          (If ((not priv)                 ; Not privileged.
            (! Right)                     ; Ask for the privilege.
            (! Right)))                   ; Got the privilege.
          (? Serv)                        ; Grant access to master.
          (? Serv)                        ; Master is finished.
          (Set priv True))                ; Remember privilege.
        ((True (? Left))                  ; Request for privilege.
          (If ((not priv)                 ; Not privileged.
            (! Right)                     ; Ask for the privilege.
            (! Right)))                   ; Got the privilege.
          (? Left)                        ; Pass privilege left.
          (Set priv False))))))           ; Forget privilege.
```

Figure 5.36: Code for Server Version 1

the privilege to this server and so this server now owns the privilege. This synchronization corresponds to the two inputs on the *Left* channel in the other alternative body. The same argument explains the multiple output synchronizations on the *Serv* channel.

Recall that synchronization is a symmetric form of communication. Because no data is passed, it makes no difference which process executes an input and which process executes an output. The only function of the communication is to synchronize the two participating processes. The fact that OCCAM channels are defined to be uni-directional dictates that the communicating processes must agree on the "direction" of the communication, but for simple synchronization, this choice may be made arbitrarily. Remember also that OCCAM semantics require unbuffered synchronization. When one process sends a synchronization signal it must wait for the other process to reply. The single exception to this symmetry is in guarding. OCCAM allows guards only on input channels. Any guarded communication, whether data is passed or not, must be on an input channel.

Initial Translation into a Circuit The initial translation of the OCCAM program in Figure 5.36 into a circuit is shown in Figure 5.38. This circuit has already been cleaned up using some simple housekeeping transformations. These improvements include removing single-client call modules, removing select modules with constant select inputs, and removing single-input XOR gates.

```
  ⋮
(! Serv)                ;request access
(! Serv)                ;access is granted when this communication
                        ; synchronizes with the server
  ...                   ;use resource
(! Serv)                ;signal completion
  ⋮
```

Figure 5.37: Code in Master that Communicates with Server Version 1

The alternative construct in the code has been replaced by the ring of Q-select modules at the top of the figure. The guards are tested by the Q-select modules and when there is input available at one of the *Left* or *Serv* channels then that alternative is chosen. According to the semantics of OCCAM, the first thing that an alternative guarded by an input primitive does is to complete that communication. After completing the communication the body of the alternative is executed.

Notice that the setting and resetting of the privilege variable can be done in parallel with the communications in each of the alternative bodies. This is easily expressed in OCCAM using the *Par* construct and results in a circuit with two additional C-elements to signal the completion of the subprocesses, but is otherwise identical to the circuit in Figure 5.38.

Circuit Transformation and Improvement The first transformation is to replace channels that are used multiple times with multiple channels. This is a difficult transformation to apply in the general case because it is often unclear that three uses of a single channel in one process will correspond to separate uses of that channel in the connecting process. In this case, however, the use of the channels is easily understood and the *Left* and *Right* channels each become two channels, and the *Serv* channel becomes three. This transformation eliminates the relatively expensive Call element at the cost of introducing a new C-element for each new channel.

The result is a server with three channels connected to the master and two channels each connecting to the left and right servers in the ring. These new channels can be thought of as *Req*, *Grant* and *Done* channels to the master, and *Req-Priv* and *Ack-Priv* channels to the other servers on the right and left. In fact, the code could easily be modified to reflect this choice of using multiple channels rather than using the same channel multiple times. One advantage of this whole technique is that the user can, by modifying the source code, change the resulting circuit to experiment with different algorithms. The modified code reflecting this multiple channel approach is shown in Figure 5.39. This code also splits the

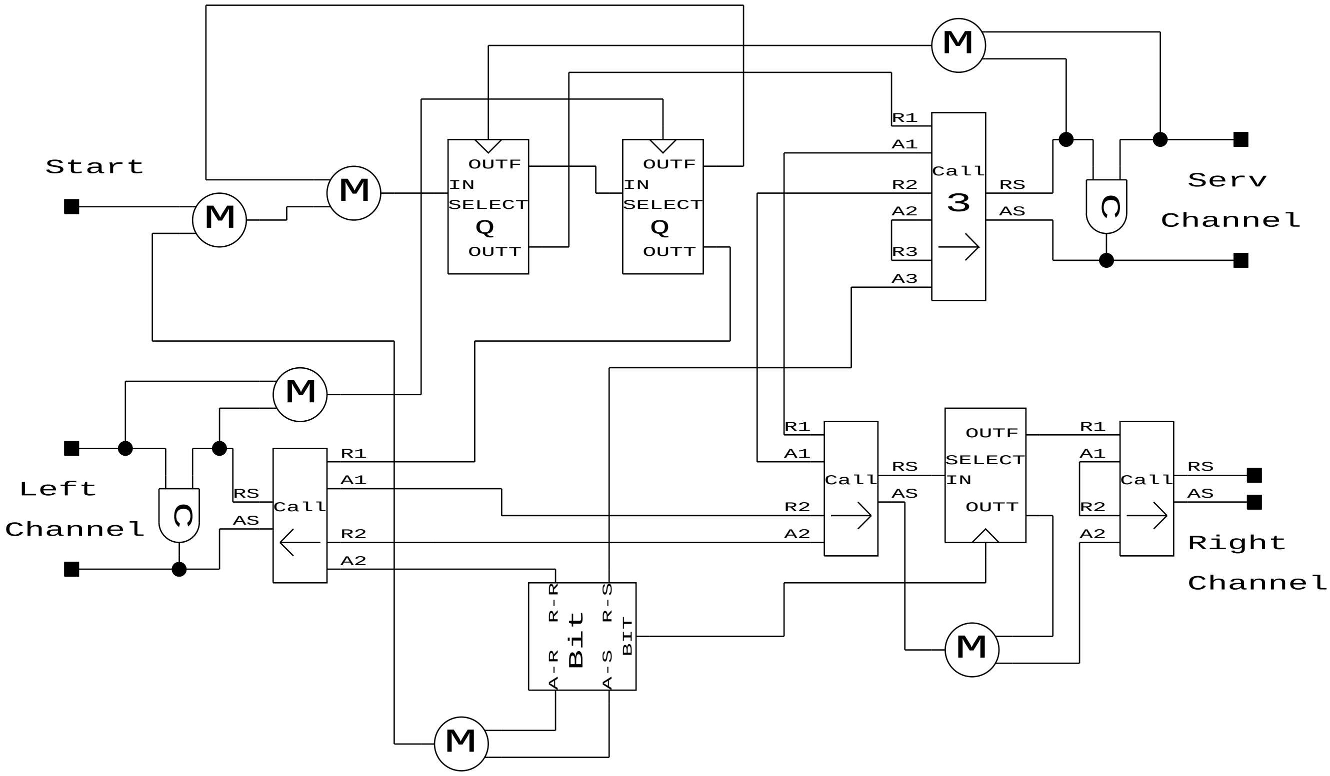

Figure 5.38: Initial Circuit for Server 1

```
(Process server1 ((Chan Req Grant Done     ; Macro to implement the server
              Req-Priv-Left Ack-Priv-Left Req-Priv-Right Ack-Priv-Right))
  (Seq ((Var priv))                        ; Define local variable to hold privilege.
    (While True                            ; Repeat forever.
      (Alt                                 ; choose an alternative
        ((True (? Req))                    ; Request from master.
          (get-privilege)                  ; Call macro to get privilege
          (? Grant)                        ; Grant access to master.
          (? Done)                         ; Master is finished.
          (Set priv True))                 ; Remember privilege.
        ((True (? Req-Priv-Left))          ; Request for privilege.
          (get-privilege)                  ; Call macro to get privilege
          (? Ack-Priv-Left)                ; Pass privilege left.
          (Set priv False))))))            ; Forget privilege.

(Process get-privilege ()                  ; Macro to get privilege from the right
  (If ((not priv)                          ; Only if not privileged.
    (! Req-Priv-Right)                     ; Ask for the privilege.
    (! Ack-Priv-Right))))                  ; Received the privilege.
```

Figure 5.39: Modified Code for Server Version 1

process of getting the privilege from the right into a separate macro process. This does not change the implementation of the circuit because the macro is simply substituted into the server code in place of the *get-privilege* statement.

In general, the resulting circuit uses fewer modules if channels that are used for specific purposes are made into separate channels. The trade off is that separate channels may use more wires than a single channel that is used for multiple purposes. The programmer is free to use whichever method is appropriate for the application.

The next transformation replaces the Q-select modules with a transition arbiter. The ring of Q-select modules is a general and extensible transformation of the alternative construct that unfortunately is active even in the absence of requests. In a case where the guard will never be revoked once it is asserted then the active Q-select ring can be replaced by a transition arbiter. In this case the incoming request from the channel used as the guard can be connected directly to the request input of the transition arbiter. The circuit for the server after these improvements are applied is shown in Figure 5.40.

The next set of transformations become available only after the server is connected to its master and to the other servers. Once connected, this is an example of the type

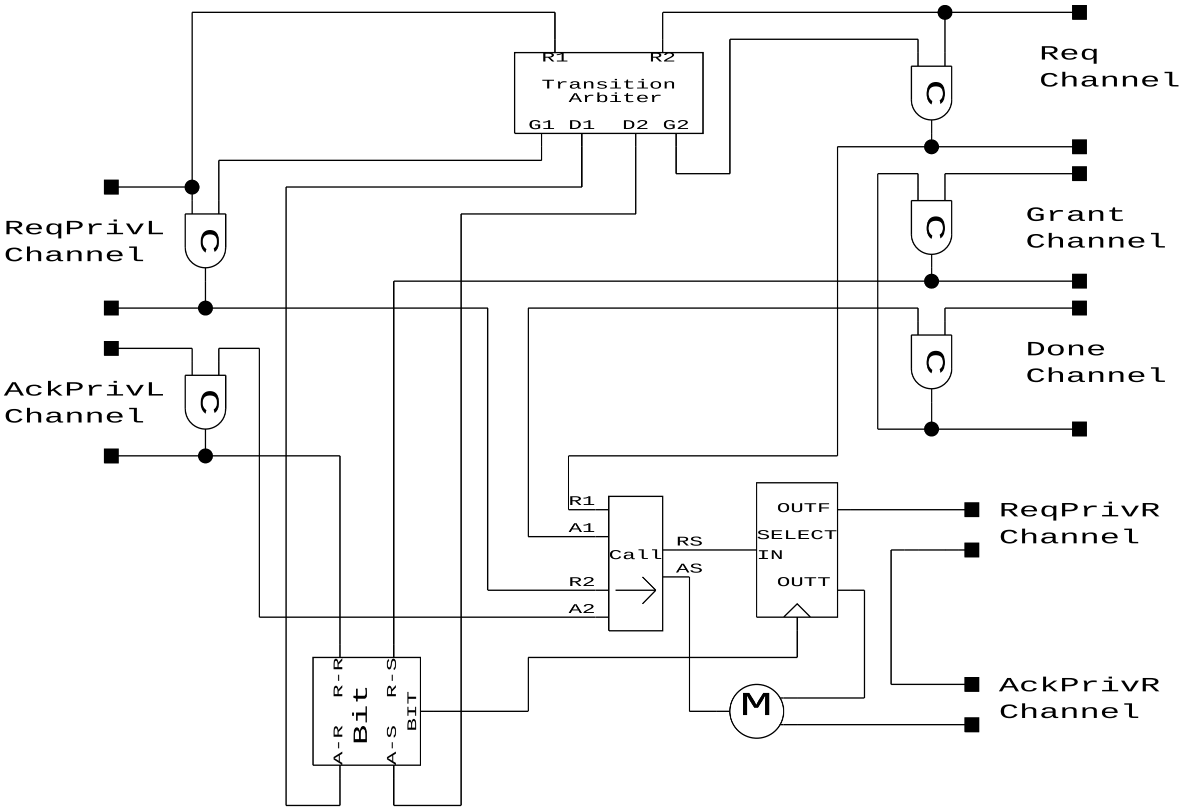

Figure 5.40: Circuit for Server 1 After Initial Transformations

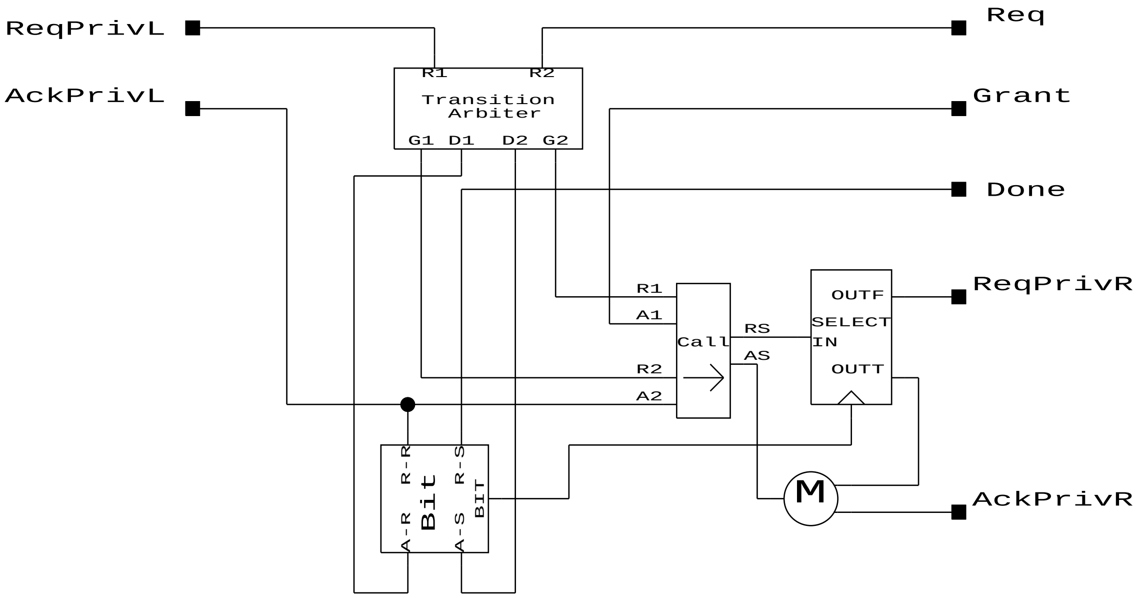

Figure 5.41: Final Circuit for Server 1

of master-slave circuit that was shown in Section 5.2.2. In the context of the assembled system, the synchronizing C-elements can be eliminated by applying C-element elimination transformations as shown in Figures 5.19 and 5.20. As in that example, in addition to reducing the number of C-elements in the circuit these optimizations also reduce the two-wire control interface of the channels to a single wire. In this case the sequential nature of the synchronizing communication between the client and server processes is reflected in the final optimized circuit. Control is passed from client to server and back on a single wire rather than the more general form of communication where either process can arrive at the synchronization point first. The key is to keep the symmetry of communication intact far enough along in the translation process to allow this sort of optimization.

Applying these transformations results in the final circuit shown in Figure 5.41. This circuit can be compared to Figure 7 in [55]. Also see a discussion in [24].

It is interesting to note that the interface from the master to the server has been reduced to exactly the same interface as to the simple transition arbiter in Figure 5.34. This is reflected not only in the *Req*, *Grant*, and *Done* style of communication, but also in that the communication has been reduced from a pair of wires in a channel to a single wire for each of these signals. The complicated channel interface between the servers is also reduced to single wires: one sending a signal to the left to request the privilege, and one returning to the right to pass the privilege to the next server.

Notice that because the privilege bit is a bundled data bit, the circuit is not completely delay-insensitive. The bundling constraint must be satisfied with respect to the privilege bit and the select module that uses that bit. This bundling constraint is easily satisfied for a circuit of this size, but the circuit could be made delay-insensitive by using a dual-rail encoded signal in place of the Boolean privilege bit at the expense of a small number of extra wires.

Distributed Mutual Exclusion - Version 2

OCCAM **Source Code** The main idea in the preceding example is that the privilege is passed around the server ring only when a server requests it from the other servers. However, another algorithm to implement this sort of distributed arbitration in a server process is to let the privilege continuously circulate around the ring of servers. When a master makes a request of its server, the server waits for the privilege to come around and then holds the privilege while the master uses the shared resource. If the server receives the privilege without a pending request from the master, the server simply passes the privilege on around the ring.

The problem with this implementation is that it can potentially exhibit livelock. For example, If the *Alt* process always chooses to pass the privilege, the server will never capture the privilege and will never grant access to its master. The previous algorithm does not exhibit this pathological property. As a practical matter, however, this will not be a problem. The *Lex-Alt* implementation specified in the following code will check its guards in lexical order, and if the first guard is true, it will eventually be chosen given the circuit implementations described previously.

The OCCAM code for this new version of the server is shown in Figure 5.42. Notice that the interface to the master is two channels in this example: a *Req* channel to request and receive access, and a *Done* channel to report that it is finished. The *Req* channel serves a dual purpose. The client process synchronizes on the *Req* channel to request access to the shared resource. The completion of that synchronization by the server is the grant from the server. That is, the client requests synchronization and then waits. When that synchronization is completed by the server, the client has received the grant and has access to the shared resource.

The server communicates to the other servers in the ring through the *PrivIn* and *PrivOut* channels in much the same way. Completing a synchronization in the *PrivIn* channel implies that that server now holds the privilege and may grant access to its client. If the client is not requesting the privilege, or if the client is finished with the privilege, the server passes the privilege to the next server by synchronizing on the *PrivOut* channel. The code used by the master is shown in Figure 5.43

The operation of the server is to wait for the privilege on the *PrivIn* channel. Once the server holds the privilege it checks for an outstanding request from the master on the *Req*

```
(Process server2 ((CHAN Req Done Privin Privout))
  (While True                       ;do forever...
    (? Privin)                      ;wait for privilege
    (Lex-Alt                        ;lexically prioritized alternative
      ((True (? Req))               ;look for Req, give Grant if found
        (? Done))                   ;wait for Done
      ((True (Skip))                ;no Req event
        (Skip)))                    ;just return
    (! Privout)))                   ;send privilege to next server
```

Figure 5.42: OCCAM Program for Server 2

```
      ⋮
    (! Req)              ;request access
      ...                ;use resource
    (! Done)             ;signal completion
      ⋮
```

Figure 5.43: Code in Master that Communicates with Server 2

channel. If there is a request it completes that communication which allows the master to execute in the shared resource, and waits for the master to signal on the *Done* channel that it is finished. Again, because there is no data passed, the communications are reduced to synchronization.

Because the privilege is passed continuously in the ring and never stored as a value in a register, there must be some way to ensure that the privilege will start circulating after an initialization. Rather than introduce an asymmetry by making one of the servers different, an extra process may be added to the server ring to start the privilege around the ring. This process simply synchronizes on its *PrivOut* channel after a clear, and then passes subsequent communications from *PrivIn* to *PrivOut*. The code for such a starter process is shown in Figure 5.44.

Note that the code in Figure 5.42 could be written more naturally using a conditional expression rather than a *Lex-Alt* statement. The purpose of the *Lex-Alt* is to check if there is a signal on the *Req* channel. If there is such a request, the process does one thing, and if there is no such request, the second alternative, which has a constant *True* guard, is selected and the *Lex-Alt* terminates. However, the semantics of this version of OCCAM does not allow an input guard in a conditional statement so this more cumbersome syntax is required.

```
(Process starter ((Chan PrivIn PrivOut))
   (While True                  ; do forever
      (! PrivOut)               ; first send out the privilege
      (? PrivIn)))              ; then wait to get it back
```

Figure 5.44: A Process to Start the Privilege Around the Ring

Initial Translation into a Circuit The initial translation of the OCCAM program in Figure 5.42 into a circuit is shown in Figure 5.45. Because the circuit is small, the housekeeping optimizations mentioned in section 5.3.2 have not already been applied. In this circuit the alternative construct is implemented as a ring of Q-select modules because the server code specifies a *Lex-Alt* or lexically prioritized alternative. In this form of alternative construct the guards are checked in the lexical order in which they appear in the code. This is important because the second alternative has a constant *True* guard and will always be able to execute so it must be checked after the first guard is checked.

In this circuit the presence of the privilege, indicated by communication on the *PrivIn* channel, causes the alternative circuit to check for a pending request from the master. If no request is pending, the second alternative will be chosen and the privilege will be passed out the *PrivOut* channel. If there is a pending request, the communication on *Req* will be completed, granting access to the master, and the circuit will wait for new communication on the *Done* channel before passing the privilege out.

Circuit Transformation and Improvement The first optimizations are simple replacements of superfluous circuit modules like the single-client call modules which may be removed. The select module with the constant *True* select input (resulting from the *(While True $\cdots$)* loop) may be eliminated, as well as the Q-select module with the same constant select input. This Q-select corresponds to the second alternative in the alternative construct and will always fire if the first alternative is not ready. Also, an XOR gate with one input connected to the *Start* signal may, in this case, be replaced by an inverter because the only effect is to ready the C-element for firing on the next change of the other input. The resulting circuit is shown in Figure 5.46.

As the server circuits are connected to their master circuit, the C-elements on the *Req* and *Done* channels may be removed following the same reasoning as in the first example. This leaves the C-element for the *PrivIn* channel to be considered.

Figure 5.47 shows how the servers are connected when they are strung together in a ring with the starter process. In this case the body of the server process is contained in the circle marked *Body*. This configuration of C-elements may be reduced to a simple wire from the output of one process to the input of another with the inverter of the starter process providing

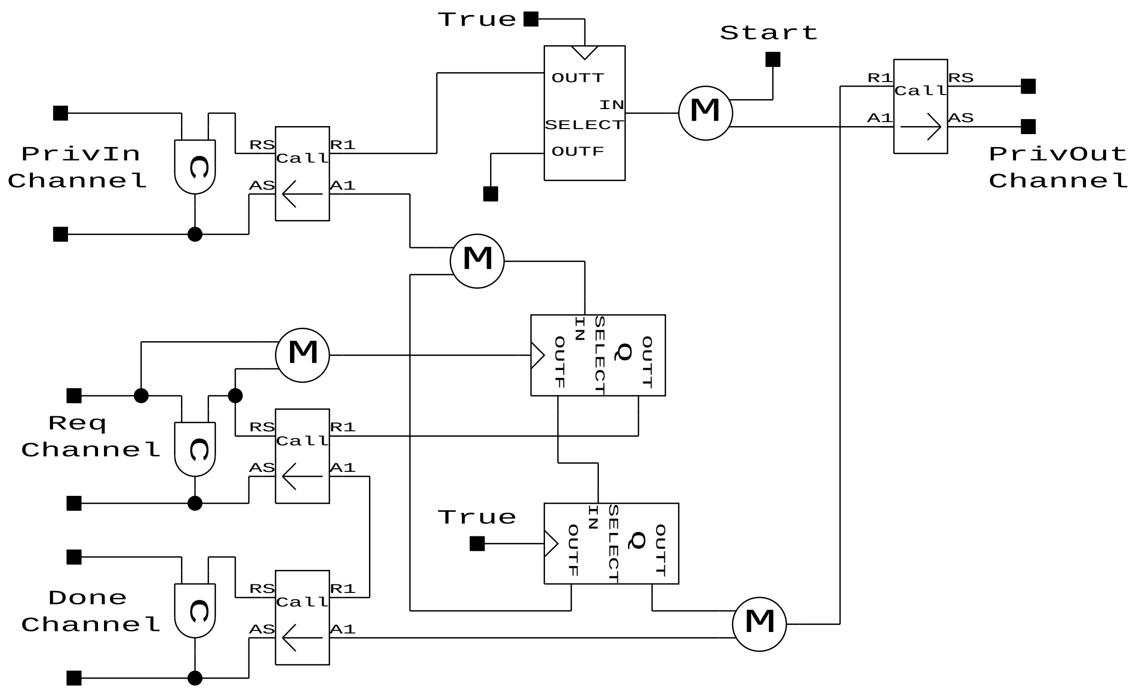

Figure 5.45: Initial Circuit for Server 2

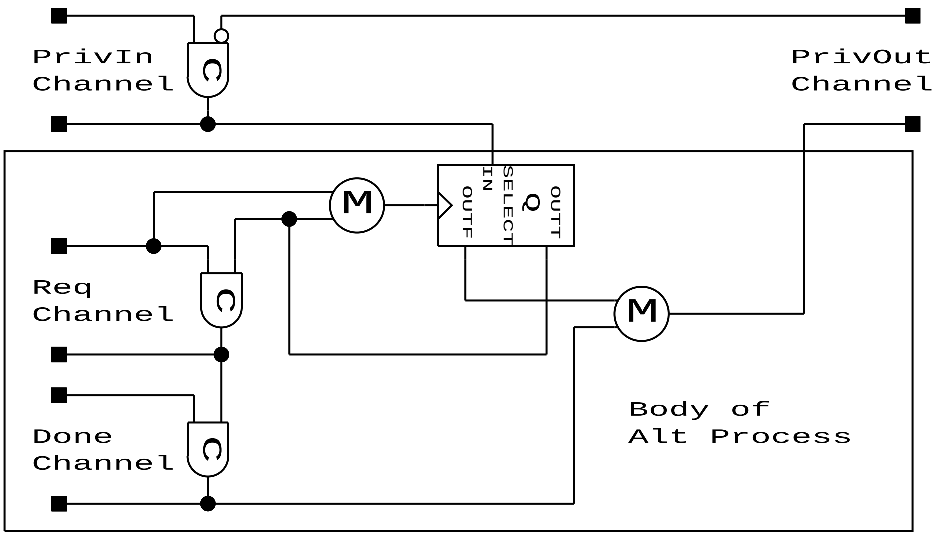

Figure 5.46: Circuit from Example 2 After Initial Transformation

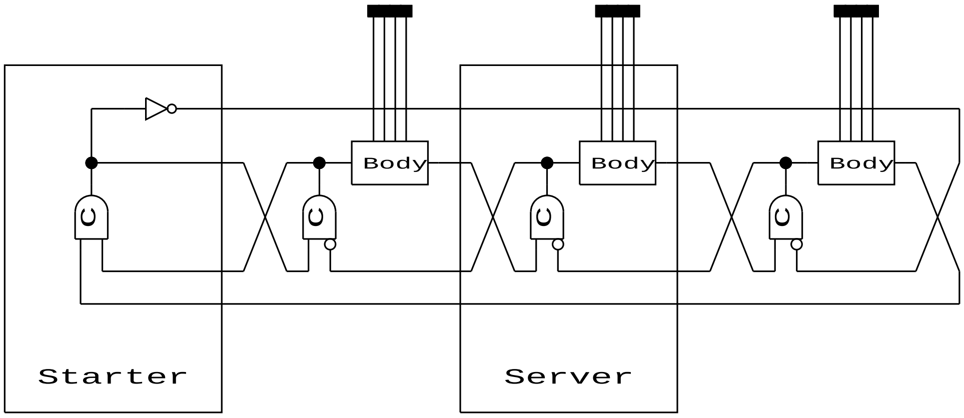

Figure 5.47: A Ring of Server Processes

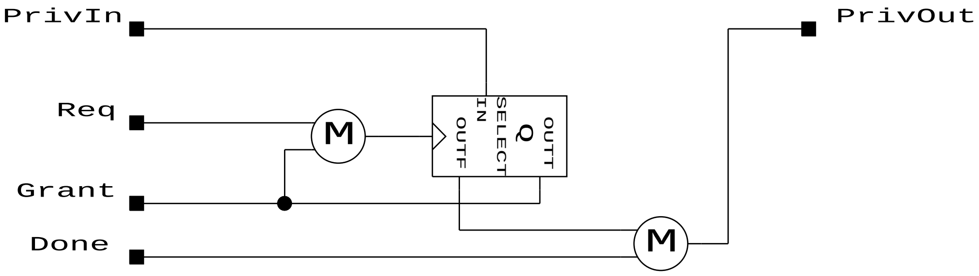

Figure 5.48: Final Circuit for Server 2

the inversion necessary to keep the ring oscillating. Because the function of the original collection of C-elements was to make sure the server processes received the privilege one at a time in sequence around the ring, it is encouraging that the transformations reduce this control to a single wire forcing this sequence. The final circuit for a single server process is shown in Figure 5.48.

Note that again, as in the preceding example, the master-server interface has reduced to the same three signal wires found in a simple arbiter. This time the connections to the other servers are reduced to a single input wire and single output wire that continuously pass the privilege around the ring of servers.

5.4 Conclusions

These examples illustrate the real strength of the translation and transformation methods. An efficient circuit is obtained from a straightforward OCCAM specification. In fact in all of these cases, the circuits that result from the translation of OCCAM code are as good as, and in many cases exactly the same as, those circuits implemented by hand to perform the same function.

For example, consider the design of a centralized n-way arbiter. One way to implement such an arbiter is with an n-way OCCAM alternative construct. This alternative construct can be constructed as a circuit using a ring of Q-selects where each Q-select checks one guard in sequence as shown in Figure 5.11. The n processes desiring service send requests to the input channels used as guards of the alternative construct. Once the request is seen as a guard on that channel, it can eventually be granted access to the shared resource.

This behavior may be written down as a collection of OCCAM processes as in the previous example. This distributes the arbiter into a collection of server processes rather than collecting them into central arbiter. If this collection of processes is compiled using the techniques presented here, the resulting circuit reduces to exactly the same configuration as in the centralized case, except that the Q-selects are now distributed throughout the system and associated with each master process. The result is a collection of Q-select modules each deciding if its guard is true before passing control on to the next Q-select.

It is also noteworthy that the communications between the master process and the server process reduce, through the use of the optimizing transformations, to exactly the same req/grant/done interface used by a standard arbiter. The extra wires are optimized away and a three-wire interface remains. The communication between the servers is likewise reduced.